PanParallacea:
a System for Debugging and Monitoring Parallel Programs

Petter Moe

9 July 1992
Acknowledgements

My supervisor, Prof. Kjell Bratbergsengen has inspired and supported my own and the hypercube groups work, he can not be thanked enough! Also, my friends and colleagues, Ståle Deraas, Svein-Olaf Hvasshovd, Eirik Knutsen, Dr. Lasse Natvig, Kirsten Springer, Dr. Tore Sæter, and Øystein Torbjørnsen have through their comments and constructive criticism contributed greatly to this thesis. The hypercube group and the department also deserve comment for providing an interesting and evolving surrounding. I thank you all!

I would also like to thank my father, through his biweekly enquiries about the state of my work, for keeping my sense of urgency intact, and my mother for trying to restrain him.
Preface

This thesis is concerned with instrumentation and debugging of parallel computers.

The layout of the thesis is depicted graphically in Figure 0.1. Note that the chapters “Introduction” and “Conclusion” are connected to most other chapters, and are thus for simplicity shown on their own.

The titles of the chapters are given as normal text in each box in the figure, together with the chapter number. The arrows indicate subjects depending on other subjects, the chapters in dashed line boxes are appendices. The text in italics outside some boxes gives an informal indication of the content.

![Diagram](image)

Figure 0.1: The Plan of the Thesis

The first part is called “Parallel Programming”. The purpose of this part is first to define a context for the reader. We initially discuss some relevant underlying
topics of parallel programming. Subsequently we will look at the different reasons for using parallel computers, and how these reasons affect our use of them. We will argue that for users turning to the often confusing concept of parallel computing, speed of computation is a very important issue. With that in mind it is clear that a debugging system must not only address the logical correctness of a program, but also its execution characteristics.

The very first question to ask of oneself when starting to work on a debugging system is: “What is a bug?” i.e. what phenomena one wants to pinpoint and examine. We therefore discuss the various problems encountered in parallel programming, as mentioned in the literature, and conclude that most problems in parallel programming arise from three basic problems: choice and enforcement of event atomicity, implicit or explicit expectancies about event ordering, and unforeseen cumulative resource consumption.

There are many schools of parallel programming active today. Each method has some advantages, unfortunately most also have disadvantages. It is however clear that the problems encountered during parallel program development and execution depend heavily on the programming model in use. We have therefore included an in-depth discussion of many popular programming models, and tried to relate them to parallel programming problems. We consider this work as novel, and an essential part and foundation of this thesis.

The second part is called “The World at Present”. We thoroughly discusses the different aspects of parallelism in the context of how to monitor performance, isolate errors, and remove bugs. We then describes the state of the art of current monitoring and debugging systems on parallel computers. We also examine the expressed goals and motivation of their creators. This part argues that it is sometimes not preferable to keep debugging and monitoring apart, as their methods and goals are often closely connected, not to say intertwined.

The third part “HC-2\textsuperscript{d} and debugging” concerns work within the Hypercube project at the Norwegian Institute of Technology. We describe the HC-2\textsuperscript{d} range of prototype computers, and their programming environment. We then continue by describing the observed shortcomings of the current programming environment, and discuss what tools this environment should be extended by.

This part is concluded by the design and implementation of a combined monitoring and debugging system, PanParallacea. Out of this work comes the main contribution of this work: a completely new way of programming parallel debugging, based on the the nodeset, the timestep, the debugger state, and the integration of debugging and performance monitoring.

The three appendices contain issues related to different parts of the thesis. Appendix A shows how other environments present information about the execution of programs. Appendix B describes another tool for understanding parallel systems, a system for simulating a parallel computer. Appendix C discusses the programming model of the HC-2\textsuperscript{d}.
Contents

I Parallel Programming 1

1 PanParallacea 3

1.1 Hints to the Reader ........................................... 4
1.2 Basic Concepts .................................................. 4
1.3 Parallel Computers .............................................. 6
1.4 Tightly vs. Loosely Coupled .................................. 9
1.5 Speedup .......................................................... 9
1.6 Environments .................................................... 10
1.7 Models .......................................................... 10
1.8 Why Leave the Virtual Von Neumann Computer? .......... 12
1.9 Why Parallel? .................................................... 13
1.10 Status and Future of Parallel Programming. ............... 17

2 Problems on Parallel Computers 21

2.1 Functional Bugs ................................................ 25
2.2 Performance Bugs .............................................. 27

3 Implicit Parallelism 33

3.1 Debugging with Implicit Parallelism .......................... 34
3.2 Optimising with Implicit Parallelism ........................ 34
3.3 Automatic Decomposition ..................................... 35
3.4 Hardware ......................................................... 37
3.5 Functional Programming ........................................ 37
3.6 Relational Programming ....................................... 39
4 Explicit Parallelism

4.1 Programming Mechanisms ........................................ 43
4.2 Transactions ......................................................... 47
4.3 Guardians ............................................................. 48
4.4 Distributed Systems .................................................. 49
4.5 Parallel Operating Systems .......................................... 53
4.6 Data-Parallel Programming ......................................... 58
4.7 MIMD Data Parallelism .............................................. 60
4.8 Communicating Sequential Processes .............................. 60
4.9 Ada ................................................................. 62
4.10 Cantor ............................................................. 63
4.11 Problem-Domain Structuring ...................................... 63
4.12 The Object-Oriented Approach ................................... 64
4.13 Delayed Mapping .................................................... 67

5 Casual Parallelism ...................................................... 71

5.1 Debugging and Optimising .......................................... 71
5.2 Bucket Programming ................................................ 72
5.3 Strand-88 ........................................................... 76
5.4 Bilingual Programming .............................................. 77

II The World at Present .................................................. 79

6 Instrumentation of Multicomputer Systems ....................... 83

6.1 What Parallel Programmers Need ................................. 85
6.2 What is Monitoring? ............................................... 86
6.3 What to Monitor? .................................................... 88
6.4 How to Collect Information ........................................ 89
6.5 How to Display Information ....................................... 91

7 Current Parallel Monitoring Systems .............................. 93

7.1 Observer ............................................................ 93
## CONTENTS

7.2 MAD .................................................. 94
7.3 IPS .................................................. 94
7.4 PIE .................................................. 97
7.5 TOPSYS ............................................. 98
7.6 Faust ............................................... 101
7.7 SIMPLE ............................................. 101
7.8 PEM ............................................... 102
7.9 ELAN ............................................... 102
7.10 Express .......................................... 103
7.11 Others .......................................... 104

8 Parallel Debugging ................................... 107

8.1 Debug Predicates .................................. 109
8.2 Parallel Debugging Problems .................... 109
8.3 Timing ............................................ 113

9 Current Parallel Debuggers ......................... 115

9.1 Remote Debuggers .................................. 115
9.2 Breakpoint Debuggers ............................. 116
9.3 Event-based Debuggers ............................ 118
9.4 Parasight ......................................... 127
9.5 Others ............................................. 128

10 Related Tools ........................................ 129

10.1 Logic Analysers ................................... 129
10.2 Simulators ........................................ 129

III HC – 2nd and debugging ............................. 131

11 Instrumentation and Debugging on HC16 .......... 135

11.1 Terminology ....................................... 135
11.2 Projects and Prototypes ......................... 136
11.3 TorOS ............................................. 138
13 Debugging Requirements 149
13.1 User Traces ................................. 149
13.2 Stopping Nodes .............................. 150
13.3 Inspection .................................. 150
13.4 Tracing & Replay ............................ 151
13.5 Debugging Performance ................. 152
13.6 Process Integration ....................... 153
13.7 Stepping .................................. 155
13.8 Parts of the Debug Controller .......... 157

14 Debug Control Language 161
14.1 Time ....................................... 162
14.2 Predicates ................................. 162
14.3 NODESET Operations ..................... 164
14.4 Debug Script Operations ................. 164
14.5 Implementation ............................ 166

15 Examples of Use of PanParallacea 169

16 Conversational Debug Interface 173
16.1 Commands ................................. 174

17 The User Interface .......................... 175

18 Implementation Issues 179
18.1 Aspects Left Out ......................... 179
18.2 System Description ....................... 180
CONTENTS

18.3 Node ......................................................... 182
18.4 Host ......................................................... 187
18.5 Controller .................................................... 187

19 Conclusions .................................................. 193
19.1 Contributions .............................................. 193
19.2 Implementation Status .................................... 194
19.3 Evaluation .................................................. 194
19.4 Further Research .......................................... 195

IV Appendices .................................................. 197

A Example Output of Utilities .................................. 199
A.1 Tcov Example .............................................. 199
A.2 Prof Example .............................................. 201
A.3 Gprof Example .............................................. 202
A.4 Xfermon Example .......................................... 204

B PPS: Portable Pool Simulation ................................. 205
B.1 Introduction ................................................. 206
B.2 Goals ....................................................... 206
B.3 Simulation Approach ....................................... 207
B.4 Implementation Basis ....................................... 210
B.5 Inherent Inaccuracies ....................................... 211
B.6 Description of Output ..................................... 213
B.7 Goal Fulfillment ............................................ 214
B.8 Conclusions and Further Work ............................. 215
B.9 Acknowledgements ......................................... 215

C VNM: Virtual Neighbour-shared Memory .................... 217
C.1 The Hypercube-16 Project .................................. 218
C.2 Communication Hardware and Programming Models ....... 218
C.3 Pairwise Shared RAM ...................................... 220
C.4 Virtual Neighbour-Shared Memory ...................................... 223
C.5 Conclusions and Further Work ........................................... 225
Part I

Parallel Programming
Chapter 1

PanParallacea

The title of the thesis is also the name of a system which provides instrumentation and debugging for multicomputers of the MIMD\textsuperscript{1} persuasion. The name is inspired from the latin panacea, a "cure for all ailments." Our system thus claims to be a cure for all parallel problems. We will proceed to show that this is not entirely true.

Parallel instrumentation and debugging is the theme of this thesis. The author believes that performance is one of the major current motivations for using parallel computers, and therefore failing to deliver performance is a problem, which we call a performance bug, as opposed to functional bugs.

Since we talk about debuggers for parallel system quite frequently, we have taken to using the term parallel debuggers. This term is not exact, the debuggers themselves are not parallel, but sounds better and is easier to pronounce than "debuggers for parallel systems". Monitoring systems for parallel computers have been "reduced" to parallel monitors for the same reason. May the purists forgive us for these simplifications; we think they go some way towards improving readability without creating too much confusion.

If the field of parallel computing is still young, the field of debugging for parallel systems is still in its infancy. We will therefore start with a discussion of some subjects which form the basis for the parallel debugging, such as parallel computers and parallel programming. We lead up to the different current forms of parallel programming, and look into how debugging fits into each of these forms.

We think our taxonomy of parallel models is novel, and the evaluation and comparison of different approaches to parallelism is one part of the contribution to the field contained in this thesis. In addition to this, the analysis and design of the parallel debugging and instrumentation environment is novel work. The designed system is partially implemented, the instrumentation part is in use, and parts of the debugging unit is in an early prototype state.

\textsuperscript{1}Multiple Instruction stream, Multiple Data stream, to be discussed shortly.
1.1 Hints to the Reader

Full references are given in the bibliography near the end of the thesis. Names of products and programs are set in bold face, italics is used to emphasise text. Program text looks like this:

program text example.

1.2 Basic Concepts

To start of with, we need to clarify some of the terms used in this text. When we use the term computer, we have no abstract computing device in mind, rather one of the actual computers which an ever increasing part of humanity have under their desk or in a machine room. A parallel computer\footnote{Garcia-Molina and Germano [GGK84] denotes a collection of communicating and cooperating processors working towards a joint goal a distributed computing system, this definition gives a superset of our parallel computers.} has more than one processor; we will mention various classes below.

Jones and Schwarz [JS80] distinguish between problems/applications and their solutions, where solution is the “strategy for solving a problem”, and can be realised by a set of slightly different implementations. We adhere to this terminology, with the exception that we use problem about difficulties and complications whenever such occur.

Different authors use different terms for denoting units of execution. In this text, we have tried to use the term process to denote one active thread of execution, living alone in one address space. Where several activities take place in the same address space, i.e. use the same global variables, we call them threads. In the case of threads, the code and data segments are shared, only the stack is private to each thread. Other authors call this light-weight processes. In our terminology, a job is a collection of threads and their data, a job with only one thread is also a process. Our jobs are by others sometimes called multi-thread processes.

Araki et al [AFC91] define:

“[.] error means the difference between a program and its specification – the difference between the behaviour requested by the specification and the behaviour performed by the program – and bug means the cause of an error.”

We have not made this distinction throughout the text, hopefully the context will make clear what we mean.

Per Brinch Hansen [Bri73] gives us some useful definitions:
Processes are called \textit{concurrent} if their execution overlap in time. More precisely, two processes are concurrent if the \textit{first} operation of one process starts before the \textit{last} operation of the other process ends. The programming techniques used to control concurrent processes are called \textit{multiprogramming}.

We use the term \textit{parallel programming} as meaning all programming techniques used to control concurrent processes on multiple processing units, cooperating to solve some problem. This encompasses implicit and explicit parallelism in programs.

Brinch Hansen points to the two vital properties of sequential programs:

- The effect of a sequential program is independent of its speed of execution.
- A sequential program delivers the same result each time it is executed with a given set of input data\textsuperscript{3}.

These properties do not only belong to \textit{sequential} programs, they belong to \textit{deterministic} programs, and Brinch Hansen’s observation is really that deterministic programs have “nice” properties.

We can split the set of all programming systems into those which are deterministic, and those which are not. There are parallel programs in both groups. Non-deterministic sequential programs are difficult to find; we observe that sequential programs seem to be a subclass of the deterministic programs\textsuperscript{4}. As long as this stands, Brinch Hansen’s observation is correct, although too restricted. Were we to find non-deterministic sequential programming systems, we would have to see the question of parallelism versus sequentiality as an orthogonal issue to that of systems being deterministic, and Brinch Hansen’s claim would only be correct for deterministic programs.

A deterministic system is “nice”, and that a system is sequential seems to imply that it is deterministic. Thus, by transitivity of implication, sequential systems are “nice”. But \textit{parallel} systems which are deterministic qualify as “nice”; too, other parallel systems do not.

Henry Bal et al [BST89] sees three differences between distributed and sequential programming:

- the use of multiple processors,
- the cooperation among the processors, and
- the potential for partial failure.

\textsuperscript{3}This is true for programs depending only on their input data; many programs also depend on a \textit{state}. The same result should be delivered from each run with the same set of input data and the same initial state [Hor68].

\textsuperscript{4}A sequential system waiting for asynchronous input can behave according to the timing of input, and as such can be non-deterministic, we let this case rest.
Bal et al [Ibidem] also distinguish between logical and physical distribution:

"A logically distributed software system consists of multiple software processes that communicate by explicit message passing."

Most other authors call this parallelism, and let distribution indicate geographical distance. We call a logically distributed system parallel if it is also physically distributed, i.e. runs on separate processors, concurrent otherwise.

1.3 Parallel Computers

Hardware is outside the scope of this thesis. For well-written and complete descriptions of parallel and sequential computer architecture, please consult the acclaimed book by Hwang and Briggs [HB85] or the newer book by Patterson and Hennesy [PH90]. The rest of this section aims to summarise and clarify terminology so that software scientists can follow the rest of the discussion.

There are various classifications of parallel computers, one of the more well known is the so-called Flynn’s taxonomy [Fly66]. Flynn refers to data streams and instruction streams, these are the sequences of data and instructions as seen by the computer during execution. There are four groups of computer systems, differentiated by the instruction and data streams being single or multiple. SISD (single instruction, single data stream) are normal, sequential computers, although the processor may have parallel functional units. SIMD computers have multiple data streams but one instruction stream over all processors, i.e. all nodes go in step and do the same thing, on different data. Conditional statements and loops may lead to some processors being disabled for a while, a conditional statement with an else part will typically lead to the processors evaluating the conditional to True first executing and the rest being disabled, afterwards the roles are switched and the else part is executed on the previously inactive processors. MISD computers have multiple instruction streams processing the same data, this combination is rare, but exists in some database computers, where separate pattern recognition units may scan the same data, looking for different patterns [Su88, Chapter “Database Filters”]. MIMD computers are independent processors, with independent instruction streams and individual data. The memory may or may not be shared, but the access pattern is individual for each processor.

Perrott and Zarea-Alibadi [PZA86] tell us that parallel supercomputers in general fall into one of two groups:

Vector processors:

This class of computers have special operations for pipelining operations over vectors, or arrays, of numbers. They often contain multiple functional units such as multipliers and adders which can be used in parallel and where one unit can take the output from another as input. These vector operations could be
simulated by running loops on other processors, but the pipelining makes the vector approach a lot faster. Still vector processing executes within a single processor, the gain of speed lies in internal pipelining and optimisation.

**Array processors:**

Array processors (or SIMD computers) are computers where the functional unit itself is replicated. All these replicas receive commands from a common controller and execute the same operations in lockstep. Individual units may be disabled, i.e. not take part in whatever goes on. Array processors can also be described as having processing power associated with memory unit, i.e. active memory. Instead of looping over an array of variables, the array processor tells all elements to do some operations on themselves.

If these are the parallel supercomputers, what we work on are general parallel computers. It is however likely that more general machines and combinations of the above will move into the supercomputing league. When MIMD goes “super”, what we do will be “supercomputing debugging”.

Flynn [Fly66] points out that what we now call supercomputers were developed to handle problems requiring “prodigious” amounts of computing time, these computers are characterised by having a high ratio of computing power to I/O capacity. He proceeds to point out that the assumptions about this ratio are too limited, high-speed I/O should be provided. Lack of I/O bandwidth still seems to be a problem in many of the faster computers of today.

Athas and Seitz [AS88] distinguish between shared memory **multiprocessors** and distributed memory **multicomputers**, and note about the compatibility of various kinds of architecture:

“Programs written in a process model with message-passing are routinely run on multiprocessors simply by implementing the message-passing operations in the global shared memory. Programs written in a process model with shared variables can be compiled to run on multicomputers by replacing assignments to shared variables with message-passing operations.”

They also claim that multiprocessors can simulate multicomputers more effectively than vice versa. Nodes with megabytes of memory are called *medium-grain*, those with tens of kilobytes are *fine-grained*.

Jones and Schwartz [JS80] contrasts SIMD stream architectures to multiprocessors; in their notation, anything with independent processing seems to be a multiprocessor. We will stick to the terminology indicated above, where global shared memory is the distinguishing property of a multiprocessor.

Bal et al [BST89] also mention *workstation-LANs* and *workstation-WANs*, where the abbreviations LAN and WAN stand for local and wide area network, respectively. These are systems comprised of workstations or computers connected by a network, whose resources can be utilised in symphony. In general they define:
"A distributed computing system consists of multiple autonomous processors that do not share primary memory, but cooperate by sending messages over a communications network."

Please do not compare this with the definition given in Section 4.4, that would only confuse matters.

Burns [Bur89] uses the expression Local Area Multicomputer (LAM) to denote a collection of host and compute nodes\(^6\) connected by a LAN, which can be used as a heterogeneous multicomputer running one homogeneous programming environment. We discuss this further in Section 4.5.3.

Burkhart et al [BEK+88] note that parallel computers are available on three performance levels: supercomputer, mini-super/super-mini, and workstation parallel computer.

A node is the unit of replication in a multicomputer, typically one processor with its memory and devices. In some multicomputers, the nodes have no direct connection to the rest of the world, they have to communicate through one dedicated external computer. This computer is then called the host.

Peter C. Patton [Pat85] enumerates the various opportunities for parallelism in computer systems, claiming that hardware people have started on the bottom of this list moving upwards, whereas system designers and programmers typically have started on top and moved downwards:

**Job execution level:** running independent jobs on parallel computers.

**Task execution level:** tasks can be partitioned over redundant systems for fault-tolerance, or over distributed systems for locality.

**Process execution level:** paired processes on SIMD or MIMD processors cooperate on tasks, non-procedural systems often use parallelism of this kind.

**Instruction execution level:** pipelined execution and interleaved memory.

**Register-transfer level:** parallelism between hardware components, e.g. wider communication and multiple registers.

**Logic level:** parallelism on the bit level.

Pipelining is used on many different levels within modern computing systems. The basic technique is to split an operation into several independent stages, which still have to be executed after one another. If several such actions are to be executed, and the different stages are executed by different participants or units, the execution can executed in parallel. As soon as the participant executing the first stage has finished working on item one he can start working on item two, even though item one may not be finished. This is exactly the same as the assembly line principle used by Henry Ford.

\(^6\)Compute nodes are in this context typically transputers added to one computer as accelerators.
1.4 Tightly vs. Loosely Coupled

These terms seem to be somewhat confusing. We have come across several interesting definitions:

- "In a tightly coupled system, the processors share memory and a clock. In the multiprocessor systems, communication usually takes place through the shared memory. In a loosely coupled system, the processors do not share memory or a clock. [...] These systems are usually referred to as distributed systems." [PS85]

- “An intrinsic MIMD computer is tightly coupled if the degree of interactions among the processors is high. Otherwise, we consider them loosely coupled.” [HB85]

- “Traditionally, a distributed architecture in which communication is fast and reliable and where processors are physically close to one another is said to be closely coupled; systems with slow and unreliable communication between processors that are physically dispersed are termed loosely coupled.” [BST89]

Ranka et al [RWS88] call multiprocessors (parallel computers with shared memory) loosely coupled if each processor has its own memory, and these local memories together make up the shared address space, and can be accessed directly (but slower) from other processors.

It seems fair to conclude that the intended interpretation of the term “coupling” should be explained whenever the term is used.

1.5 Speedup

When discussing the performance of a program, we need some way of measuring how well it performs. Speedup is the ratio between the time needed for the most efficient\(^6\) sequential algorithm to perform a computation and the time needed to perform a computation on a parallel computer [Qui87, MS87]. If we let \(T(1)\) denote the time for the best sequential implementation of an algorithm, and \(T(N)\) the time on \(N\) processors, the speedup \(S(N)\) and the efficiency \(E(N)\) can be expressed as:

\[
S(N) = \frac{T(1)}{T(N)}
\]

\[
E(N) = \frac{S(N)}{N} = \frac{T(1)}{N \times T(N)}
\]

\(^6\)Seitz [Sei89] warns about articles where the speedup is compared with solutions other than the best sequential, he thinks they can be confusing.
If the speedup is higher than the number of processors, i.e., if the efficiency is more than one, we have superlinear speedup.

### 1.6 Environments

Dart et al [DEFH87] discuss different environments, and makes the following distinction:

"By *programming environment* we mean an environment that supports only the coding phase of the software development cycle – that is, programming-in-the-small tasks such as editing and compiling. By *software development environment* we mean an environment that augments or automates all the activities comprising the software development cycle, including programming-in-the-large tasks such as configuration management and programming-in-the-many tasks such as project and team management. We also mean an environment that supports large-scale, long-term maintenance of software."

It seems reasonable to see debugging as part of the coding phase; an environment giving the user support for coding, compilation, linking and debugging is then still a *programming environment*. Software development support is a large and fascinating area of research and development, but we consider it outside our scope and will not go into it any further in this thesis.

### 1.7 Models

To implement an application on a computer, the programmer needs a model of computation, i.e., a way to think about the world in which programs execute. Some advanced high-level styles of programming, e.g., logical and functional programming, offer good candidates. Within the realm of traditional, sequential computation, computer programmers think of single, sequential processors accessing passive memory. Boring as this model may seem, it is such a common conceptual model that it is often seen as the only way of thinking of how a computer works.

When programming explicitly parallel, this model is not satisfactory. Kung [Kun89] describes the nine following computational models for one-dimensional processor arrays which are also applicable for other architectures:

**Local Computation:**

Under this model, each node works on a separate part of the output set. No communication between participants is necessary. If the work can be partitioned reasonably, this model of computation is easy to work with and efficient.
1.7. MODELS

Domain Partition:
This model is introduced where the domain is split in subdomains with inter-domain dependencies. The nodes can do most of the work locally, typically for all of their output domain, but have to communicate with other nodes to get all the input required.

Pipeline:
Under the simple pipeline model, different nodes work on different phases of a computation. Each node gets a set of input consisting of the result of the previous phase, and computes the result of the current phase, which it then sends on to a successor.

Multifunction Pipeline:
The pipeline model can also be used where different nodes do different transformations on their respective input data. This is similar to what in general computing science is just called pipelining, e.g. the first processor could do lexical analysis of a text and pass tokens on to the next, which then parses a program.

Ring:
If the number of logical nodes in a pipeline computation exceeds the number of physical nodes available, a logical to physical node mapping is required. Grouping consecutive logical nodes onto one physical node may lead to bad performance and significant delays before the last processors gets going. Mapping in a round-robin fashion is an alternative to this, we call this the ring model.

Recursive Computation:
The model where input data is passed “inwards” in a row of processors, which send their results back to the requestor is called recursive computation. The distinguishing property of this model is the opposite direction of input versus output.

Divide-and-Conquer:
This model describes the situation where a problem can be partitioned, and communication between participants with varying relative distance is required to produce the result. A typical example of this is sorting, where one node in successive stages of the computation may talk to nodes N, N/2, N/4 down to 1 “hops” away from itself.

Query Processing:
Under this model, we imagine a database spread out over all nodes. A select request may be sent to all nodes, which process their local parts and output their results. These results may have to be processed further to get the final result.

Task Queue:
Under this model, nodes can be assigned tasks dynamically as soon as they are
available. A list of tasks is available, which nodes pick from when they finish another task.

In the list above, we miss the “competing participants”, where a set of nodes work on the same task, pursuing different strategies to find a solution, and the solution first found is taken. Jones and Schwartz [JS80] call this the Chinese Army approach.

1.8 Why Leave the Virtual Von Neumann Computer?

From the beginning of the age of computing, most of the computers available have had single processors. This has influenced the entire field of computing to the extent that the concepts and notions of the single-processor, sequential so-called Von Neumann computer have been the basis for most of the work within the field. This has changed in the recent years, parallelism and distribution have been investigated for many years, and are now increasingly utilised in commercial products. However, many of the fundamental ideas of computing are still based on the single-processor machine. Imperative languages executed statement by statement are closely linked to this machine, and still essential in the teaching of computing science.

The early computers were huge achievements of their time. Their underlying model was well defined, and fairly agreed upon. A processor had a set of registers, and some random access memory. The processor could manipulate its registers and memory pretty much as it liked. There were no other processors or processes around.

Van Horn [Hor68] describes the concept of a Virtual Computer, or Virtual Machine. This machine is an abstraction provided to the programmer by the operating system and the real computer, and concerns how the programmer accesses the resources of the machine. A programming model very similar to the Von Neumann computer has been widely used, and has proved successful. Even in many modern operating systems each process sees a linear memory space and an “individual” set of registers. Each process can behave as if it had a computer to itself, and coordination of these computers is

---

7My knowledge of war strategies of the far east is very limited, but apparently one of the distinguishing properties of the Chinese “People’s army” is a lack of technical equipment, which has lead to a willingness to tolerate an extreme loss of other resources, like humans, in their operations. This has been called the “human floodwave” approach. If we see equipment as stretching the abilities of the individual, the Chinese have used a strength in numbers to compensate for a comparative weakness in individual (technical) strength. Since the Chinese army has largely moved on foot, or on bikes, they have also been able to pass terrain where vehicles could not get through, but it is not clear to me how Jones and Schwartz could have had this in mind when referring to their tactics.

8Patterson and Hennessy [PH90, Page 23] think the efforts of J. Presper Eckert and John Mauchly were at least as important as those of John von Neumann, and that the term “von Neumann computer” should be avoided for that reason. We use it to indicate computers with single processors and linear memory, but the real innovation lay in storing programs as numbers in memory together with data, instead of implicitly by wires and switches.
handled by the operating system [PS85]. Operating system improvements like time-
sharing have typically tried not to upset this view of the computer.

It is today common to have multiple users executing on a computer at the same
time. The operating system will schedule the processes and give them access to
resources according to a priority. This is based on some formula calculating over
past or expected future usage, identity of the user, priority of the process and other
factors\(^9\). Each process will usually know little about other processors; it runs when it
can and does often not even notice the other processes. Logically seen, it is executing
alone on a simple, sequential processor. There are many good reasons for using this
simplified model, it is well known and can be modelled on various types of hardware.

If the number of processes on a parallel computer is larger than the number of avail-
able processors, the best solution may well be to let one or a set of processes live on
each processor permanently. They then execute as they would on one single proces-
sor; within the individual processors each process runs on a virtual machine. The
effect of having multiple processors would then be to increase the total throughput of
the system, hopefully proportionally to the number of processors. If the set of pro-
cesses can be partitioned in a reasonable way between the processors, and the need
of each process can be satisfied by one processor, such a simple and clean solution
may well be ideal. A computer with multiple processors does not need to duplicate all
other hardware, therefore increasing the number of processors should reduce the price
per performance. Hence the speed of the parallel system should scale superlinearly
compared with the investment in the computer.

Unfortunately, the virtual processor approach is not always adequate. Some users
want to solve large problems. Solving large problems on one processor will take a
significant amount of time. Even if the processor in question is very powerful, someone
will think of a problem so large that waiting for the solution from one processor is just
not acceptable. We will need to make several processors cooperate on one problem.
Then the processes' abstracted view of the computer gets out of touch with reality.
The efficiency of a system will often depend on how closely the abstractions used are
related to the real world. Since the virtual machine abstraction is far from reality in
the case of a multiprocessor computer, new abstractions are needed.

However much we would like to continue programming for our simple, sequential
virtual machines, a time will come when this will not be appropriate. For many
applications, that time is now.

### 1.9 Why Parallel?

There are reasons for using distributed systems, by which we mean systems of in-
dependent, autonomous computers, spread out through a building or over a larger

---

\(^9\) Clever programmers will always find out how, and try to manipulate this allocation. As an
example, on systems where user interaction increases the priority of a process, programmers are
known to introduce arbitrary, unnecessary print statements to get classified as interactive.
geographical area [BST89]. Among these reasons we find a need for utilising functionality only offered by some of the computers available in the network. In distributed database systems we often find that one application spans several locations, but that a great deal of the operations (transactions) involve only local data within each of the participating computers, this is typically the case with banking systems. Instead of shipping all requests to a remote centralised database (the headquarter), most of them can be handled at your local branch. Remote data are also available, although often at a higher cost.

Recently, we have seen the increasing trend that computer owners find it wasteful that most of their workstations sit around idly much of the time. They therefore want to utilise these wasted cycles, in effect putting their users into the jungle of a distributed computing system, or network computer.

There can also be many reasons for using a parallel computer. Some people will argue: “A parallel computer is what I have, so that is what I must use. My jobs are still small, however!” When this is the case, utilising the resources is not difficult. One sticks to the virtual machine, approach, and tries to forget the embarrassing and complicating fact that the executing processor is not the only one in the box. But as mentioned above, some people will actually need parallelism. To be more specific, some applications may need to reduce response time and others will need to increase total processing capacity, to achieve this they need multiples of the processing power offered by single instances of their processors.

Some applications will even use parallel computers because of their simplicity and elegance! A parallel implementation may (or should, at least) in many cases be easier to use for modelling some systems, since the “Real World” is inherently parallel. Hillis [Hil86] gives one example of this, where each processor in a massively parallel computer is used for modeling a single particle in a simulation of fluid flow. Fox et al. [FJL+88] is full of practical examples of parallelism in applications programming, where the parallelism of the problem is well matched with the parallelism of the solution. Among these we find problems as diverse as matrix operations, distributed simulation, and sorting.

Patton states [Pat85]:

“While the world around us works in parallel, our perception of it has been filtered through 300 years of sequential mathematics, 50 years of the theory of algorithms and 28 years of Fortran programming. Despite this, we often discover 30 percent of linear speedup surviving that three-state filter. This result shows that much of the parallelism inherent in nature has survived our sequential mathematical models and methods.”

We do of course realise that parallelism may introduce new sources of errors and confusion, indeed we find it likely that the amount of extra difficulty involved will be

---

10At least, so-called network programmers claim this to be a trend, we are not quite sure if the customers have realised this yet.
1.9. WHY PARALLEL?

related to the degree to which one wishes to exploit the system. A discussion of some of these problems appears in Section 2.

Karp [Kar87] claims that

"The point of parallel processing is to reduce the elapsed time to complete the job."

Miller and Yang [MY87] are slightly less conclusive, but still:

"An important motivation for writing parallel and distributed programs is to achieve performance speedup."

Let us leave no doubt, speed is important!

In fact, the question of why anyone would want parallelism can be reversed: "Why should anyone want to stay with sequential computers?" Parallel computers are faster than sequential ones, they have a higher performance per cost, and they can run sequential programs. One of the few legitimate reasons left for buying single-processor computers is that of cost, since sequential systems are smaller and simpler the total cost of such a system can be attractive, although the cost per performance will not be. Many people also of course wish to continue running old software to get as much as possible out of their investments, and will not switch to parallel computers before their old software can be executed on those. Related to this we find the conviction that writing software for new architectures is too difficult, therefore all new software should also be written in the old ways. As we will proceed to indicate, the use of modern programming methods may invalidate this argument.

An important question to ask, as Jones and Schwartz [JS80] do, is if there are substantial problems for which only sequential solutions are possible. They believe, and so do we, that no such problems to exist:

"Invariably, large realistic applications have a solution whose implementation can usefully employ at least some parallelism"

Some researchers consider using the extra capacity of added processors for speculative parallelism, [Jon86, Chapter 24] [Qui87, Chapter 10], meaning that each processor solves some partition of the problem, where the solution to subproblems may or may not be needed. Using this technique, processor cycles may be wasted. Too much parallelism will give waste, too little will not speed up the solution as much as wanted. Arguably, wasting some processing time may be better than idletime, but as described above idle processors could often be doing useful work for someone else. I sincerely believe that there is no such thing as "enough memory" or "enough processing power". Even as the amount of available resources increases, we will always need all that we get; we will even probably wish that we had more. Therefore we should aim to utilise what we have as well as possible!
Peter C. Patton [Pat85] discusses the development stages of multiprocessing. Originally multiprocessing was *throughput-oriented*, i.e. it was focused on increasing the total throughput and utilisation of the parts of a computer. This lead to the overlapping of I/O and computing, and to the development of parallel computers with independent processes which do not cooperate. *Availability-oriented* systems were to follow. Such systems are typically implemented by duplicating components on some level, and adding communication. This technology will develop further, up to the time of his article (1985) products existed only in special market niches. Patton notices that this market grows much faster than the computer industry as a whole, and foresees a “spirited contest for market shares in the second half of the 1990s.” What is usually called parallel processing is really *response-oriented* multiprocessing, where several processors cooperate on giving better response to a single application.

As we see, there are good reasons for using parallel computers, but it is still a question how much of the parallelism the programmer should need to see. In some systems, explicit exploitation of parallelism may be necessary, and the debugging and monitoring of such systems is the main theme of this thesis.

### 1.9.1 Fault Tolerance

A parallel system will contain several components individually prone to errors. These partial faults may look as a problem; they are also the seed of a feature. When a sequential processor crashes, its internal state is gone, and it has to be restarted from scratch. A parallel system can be designed to tolerate and compensate for individual failures, and allow the system to live on and carry on computing. Some applications may provide services so vital that they cannot be entrusted to a system prone to crashes, and will want to exploit this possibility to safe-guard against failures. This will require some degree of redundancy of data and processing power, both to detect partial errors and to correct them. In many cases the certainty\(^\text{11}\) that data will survive and be available will justify the extra cost.

Flynn [Fly66] pointed out that:

> “Complex system with very large numbers of components are naturally very difficult to maintain, unless features which provide fault location are included. If checking is not included in the hardware, then it is, of course, incumbent on the user to program a thorough check of his results. This, of course, represents an overhead which penalizes the effective performance and utilization of the equipment.”

Current research aims towards not only detecting, but also masking many kinds of errors. Still the observations hold: larger systems will have more faults, and although costly, these must be handled; preferably automatically repaired.

\(^{11}\)Certainty is here defined as the probability of failure being lower than some limit.
1.10 Status and Future of Parallel Programming.

James J. Hack [Hac89] looks at limits of parallel computing performance from a mathematical point of view, and claims that believes in “The Teraflop Computer” soon are the result of “an extremely inaccurate notion of the current status and short-term potential of parallel computing”. His observation is that the performance of top-of-the-range computers follow a curve now flattening, and that the scalability of current technologies is limited. With a little hindsight and recent developments like Intel’s Touchstone computers and the Thinking Machines CM-5, our personal feeling leads us to believe that “we ain’t seen nothing yet.” Only time can tell.

Jones and Schwarz [JS80] gave a status report on multiprocessor systems in 1980. Some of the practical details are getting somewhat dated, but it is interesting that the authors already at that time claimed that parallel programs need not be more difficult to write than sequential ones, and that waste of resources due to parallelisation need not occur. Even in 1991, neither of these claims are generally accepted.

Larry D. Wittie [Wit89] is not thrilled by writing parallel programs:

12In this specific case, a two-processor system executes at 1.6 times the speed of a single processor. The system uses checkpoints and logging of messages, in addition to some replicated server processes.
"Designing parallel and distributed algorithms is hard enough; successfully implementing them is universally painful."

But then again he discusses debugging traditional C programs augmented by send and receive calls; it seems appropriate to suggest that his problems depend on the tools as much as the task.

Athas and Seitz [AS88] are more cheery:

"In our experiments with multicomputers over the past eight years, we have repeatedly underestimated their application span and overestimated the difficulty of writing programs. In fact, for certain applications the concurrent formulation for a problem has been simpler than the sequential formulation."

Alan H. Karp [Kar87] divides parallel computers and their programming models in three: message-passing schemes, shared-memory and hybrid systems. He then proceeds to describe how the programming model influences programming, richly illustrated with coding examples. His observation is that algorithm design for shared memory systems is easy to do. Debugging, on the other side, is difficult, since the only thing to be observed is that some variable has the wrong value. Designing good algorithms for message passing systems is harder, since message traffic usually should be minimised. Karp finds debugging for message-passing architectures to be easier than for systems with shared memory, although far from easy, since the machines tend to stop at the point of error. This is an interesting observation, in our own experience message passing programs do not stop, but still the problem can often be traced back to an erroneous message, or the lack of one.

We have no personal experience with debugging shared-memory computers. The argument that shared memory is easier to program since communication is implicit in variable assignment is often used. If this property lets communication and in the next instance algorithm design get informal and unstructured, it seems only too believable that cooperating processes will run into problems.

However, hybrid systems are said to be the worst of both worlds: the errors are as hard to find as on shared memory machines, and algorithm design is as hard as on message-passing systems. Hybrid systems are possibly easier to build, though.

Gajski and Peir [GP85] split parallel programming researchers into four schools, those believing in:

**faster circuitry** with a mechanism for synchronising processes,

**optimising and vectorising** compilers,

**new parallel algorithms** through new languages

**new models of computation** e.g. data flow.
These schools (still according to the authors) have all failed to recognise:

"that VLSI technology brought us not an evolutionary step forward but a revolutionary one."

The authors focus in on three problems: partitioning each problem into tasks, scheduling each task for execution on one or more processors, and synchronising control and data flow during execution.

Jones and Schwarts [JS80] remind us that if a solution is "well-matched" with the underlying architecture, it will utilise it better. The space of parallel solutions is characterised by three dimensions:

**The computation unit:** i.e. the granularity of the processor (or process) similar to that of the solution?

**Communication patterns:** i.e. the frequency and amount of communication in the solution relevant for the architecture?

**Patterns of reference to data:** i.e. the regularity and locality of reference similar, vector processors will behave very well under some patterns of reference, but bring little under others.

Patton [Pat85] sees three basic approaches to research in parallel processing:

**Hardware-driven** where software is added afterwards, as seen fit,

**Software-driven** where a language or model is first designed, and then possibly fitted to existing hardware, and

**Application-driven** where an application is first selected. Then a model of computation is found where the problem can be solved, and soft/hard implementation is done at the end, with efficiency in mind.

Patton foresees some years of parallelising traditional algorithms with explicit methods before novel, declarative languages lead to new architectures and may harness large numbers of microprocessors.

We believe this point to be valid. Much of the current work within parallel debugging really stems from an antiquated model of computation. When we finally turn to programming tools which are on a high level also with regards to parallelism, our needs for and solutions to error location and correction will change dramatically. When that day comes, the solutions sketched in this thesis will be out of date!
Chapter 2

Problems on Parallel Computers

The field of computing science is still a young one. In some fields we have, through the outstanding insight and clarity of a few and the continuous work of many, gained a good understanding. This new understanding has subsequently allowed us to attack new problems and aim for higher goals, thus encountering new and sometimes unexpected phenomena. A large number of new uses of computers are now being discussed, under various labels. Expert systems and artificial intelligence spring to mind. In the design and realisation of parallel solutions for all these applications we encounter difficulties of many kinds. In this work we have chosen to work on problems directly related to parallelism. We will eventually focus in on parallel computers with programming models where the parallelism is explicit, the code imperative, and message-passing the means of communication, but some problems of parallel computing do not depend on these characteristics.

Many of the problems of computer science are old, well known, and independent of parallelism, they are difficult in both a sequential and a parallel setting. Some old problems even seem get worse if combined with the control of parallelism. As we will proceed to show, parallelism can often be hidden from users and programmers. Often, the real problem in that case is that shortcuts and “hacky” solutions designed for sequential computers fail to work on new, parallel computers. As an example of this, take the implementation of critical regions. The purpose is to make sure that only one process is allowed to modify a structure at a time. On a monoprocessor, this can be implemented by making sure that a process entering the critical region is allowed to execute continuously and on its own until it leaves the region, for example by raising the priority of the process to the maximum possible value or disabling interrupts. If multiple processors have access to the same memory, a process running on a different processor will not be affected by these measures, and may illegally modify the structure. The reason for this problem is that the original solution only worked in a special case; with more general and robust methods not depending on special properties of one particular architecture, the problem can still be solved. Since this problem stems from sequential thinking, it should not be ascribed to the parallelism involved.
CHAPTER 2. PROBLEMS ON PARALLEL COMPUTERS

Some languages and implementations may gain little from parallel execution, this is of course a problem if the user wants speed, but it seems fair to give blame for this to the designers or implementors of the original system instead of blaming parallelism for not delivering the goods.

It is important to point out that there are several very different reasons for problems in parallel programming:

- Some problems occur because there are independent activities taking place, we find these in multi-thread programming as well. These include ordering of events and interleaved access and update of datastructures.

- Some problems appear because the processes work in different address spaces and have to communicate explicitly, we find these in multi-process programming and distributed programming, too. When one participant wants to tell another something, it may not be ready for a message or it may want a message of a different kind. This also limits the programmer wanting to specify criteria over variables living on different nodes.

- Some problems occur because the activities take place in separate places, these problems are also to be found in distributed programming. This last group includes the problem of partial faults, i.e. that part of the system may fail without taking the entire system down. These problems were often mentioned in early literature, but seem to be less difficult now, probably physical distance is among the easier realities to hide.

Ever since the introduction of computers and programming languages, some curious minds have enquired how anyone can be assured that programs actually do what we think they do. Program verification is used as a means towards that goal, but unfortunately it seems to be quite difficult to really get to grips with what computers actually do and exactly what we expect of them. People working in this field seem to have much work ahead of them even without involving parallelism, and even more when taking it into account. Sommerville [Som89] clarifies mathematical program verification as opposed to program inspection and static program analysis. Lamport and Schneider [LS84] discuss proof systems for shared-memory communication and synchronous message passing. Chandy and Mishra [CM88] take a good look at parallel programming and correctness.

As computers get stronger and faster, they will be used to run larger programs. Many researchers and practitioners claim that software engineering related problems scale superlinearly with the size of programs. Commercial programs for parallel computers are likely to be very large and complex, so the task of getting programs correct will get ever more difficult. Programmers use advanced abstraction and programming techniques to counter this trend.

In this thesis, we try to address problems related to parallelism. Where they are not and where they can be masked, we have called the resulting debugging process trivial. We do realise that there are many problems left, in some cases unsurmountable, and
that trivialisies is not a term that will be accepted by those working on them. We use the term trivial about debugging to indicate that the problems of special interest to us, i.e. the parallel ones, are not present.

Jim Gray [Gra85] has observed that most hardware faults are soft, or transient, meaning that they disappear if the action showing the fault is repeated. On his basis of experience from the Tandem computers, he conjectures that this is also the case with software faults. The reason, we are told, is that only bugs depending critically on some very particular state of the computer can survive intense testing. We find it reassuring that software leaving Tandem is so well engineered and tested, but we are not convinced that this is the case with all computer programs available today, not even all the commercial ones. Gray draws some parallels:

"Bohrbugs, like the Bohr atom, are solid, easily detected by standard techniques, and hence boring. But Heisenbugs may elude a bugcatcher for years of execution. Indeed, the bugcatcher may perturb the situation just enough to make the Heisenbug disappear. This is analogous to the Heisenberg Uncertainty Principle in Physics."

Both McDowell and Helmbold [MH89] and LeBlanc and Mellor-Crummey [LMC87] also point out that non-deterministic programs will often alter their behaviour unpredictably when being debugged, and that parallel programs very often display some degree of non-determinism.

In order to keep some of the parameters of concurrency fixed, most multicomputers are space-shared but not time-shared. With time-sharing, communication patterns and scheduling would depend more on the program mix at any moment than on the internals of each program. Prediction of communication and task scheduling effects and working for increased resource utilisation would be a different task from what we now face. With time-sharing, minimal use of all resources is optimal, since other programs can then use any surplus capacity. Local amelioration and a good mix of programs would be the formula for good execution.

Without time-sharing, resources not utilised are just wasted. Therefore we do not blindly strive to reduce the consumption of the individual resources, we improve performance by aiming for a full and balanced use of all resources without wasting unnecessarily. As an example of this, imagine a program needing to do a lot of local computation and a lot of communication, where these activities can be performed in blocks, i.e. first computing, then communication. If executed on its own, this will lead to a waste of communication bandwidth while computing, and a waste of CPU-cycles while communicating: time used will be \( T(\text{think}) + T(\text{talk}) \). If the communication can be done in parallel with the computation, time can be reduced\(^1\)

\(^1\)The Hypercube group (described later) observed an effect similar to this in the implementation of relational join on our prototype computers [Bra87]. The processing and redistribution (communication) was overlapped with disk-operations, and the total time of the communication was practically identical to the time of the disk-operations. Parallel use of these three units gave us the processing AND communication time "for free".
towards \( \max(T(\text{think}), T(\text{talk})) \). Added work will reduce the execution time if it reduces the critical path, even if it increases other parts.

Creve Maples [Map85] mentions potential software problems in a multiprocessing environment. The parallelism inherent in the decomposition of a problem will limit the parallelism during execution. Also, the higher the number of participating processors the higher the impact of sequential code will be on the speedup. Cooperating processes have to coordinate their execution and results, this can be difficult and expensive. Lack of synchronicity and determinacy limits the programmer's control of what actually happens at any time, and improvements are hard to make. All the problems of serial code add to this. On the hardware side, Maples sees contention as limiting in systems with shared resources, less so in others, but some overhead will be present in all systems.

Lewis [Lew90] argues that parallel programming keeps all of the problems of sequential programming, but in addition gets non-determinism, race-conditions, synchronisation and scheduling problems. Tsai et al [TFC90] have experienced that synchronisation and timing errors are the most difficult to detect.

Ahuja, Carriero and Gelernter [ACG86], however, base their experience on Linda (see Section 5.2.1) and observe:

> "Parallel programming is often described as being fundamentally harder than conventional, sequential programming, but in our experience (limited so far, but growing) it isn't."

Quinn and Hatcher [QH90] claim that asynchronous algorithms are more prone to time-related errors, in comparison with synchronous computing this is undoubtedly true. Below, we have chosen to look at more specific problem instances, some of these also qualify as typical time-related errors.

Other authors agree that timing is an issue, [PS85, pp. 375]:

> "A language must provide the means to guard against time-dependent errors. Such errors can occur if several concurrent processes communicate with each other by the use of either common variables or explicit messages."

This guard should come in the form of high-level mechanisms integrated in the conceptual model of the language and programming model, and thus be enforced all through the implementation.

Many of the parallel problems occur because parallel programs are not deterministic. We would like to point out that implementations with control of the access to shared resources or data usually do not enforce determinism. Such control includes semaphores, critical regions and the like; they can enforce indivisible access\(^2\) and re-

\(^2\)As Bal et al [BTK90] point out, atomic often involves recovery, hence the term indivisible is less likely to be misinterpreted.
move some problems, but nondeterminism depends on races\textsuperscript{3}, and may still well be present.

Lewis [Lew90] observes that programs where parallelism is hand-coded are equivalent to low-level programs, since they are architecture dependent, both for correctness and performance. We agree with this statement and the implicit criticism, programmers should strive for a higher level of abstraction!

\section{2.1 Functional Bugs}

If we set aside all problems of constructing parallel algorithms and solutions, most parallel bugs have a common denominator: multiple threads of execution. The direct reason for a bug is usually one of the following:

\textbf{Event atomicity:}
An event can be for instance a data read or write, message sending, or memory allocation, the problem is that several events are related and should be executed without intervention. This encapsulation of a sequence comprising an event can be enforced, a common problem is that it is not. One could also say that this is a problem of event granularity.

\textbf{Event ordering:}
The programmer may assume a certain event order, and thus not handle events out of that order correctly. This can, but need not be related to the above problem.

\textbf{Cumulative resource consumption:}
One process will use a certain amount of all available resources. Assume that the usage is dynamic, and that the resources are finite. With several coexisting processes the cumulative resource consumption may at some unpredictable time exceed the available amount, and probably bewilder the programmer.

In the following sections we mention some problems encountered in the literature.

\subsection{2.1.1 Lost Updates}

One problem in parallel programming is that of lost updates. This appears in two ways, the first is rather low-level and occurs when two different processes or processors

\textsuperscript{3}A race occurs [MH89]:

".. whenever two activities are allowed to progress in parallel. For example, one process may attempt to write a memory location while a second process is reading from that memory cell. The second process's behavior may differ radically, depending on whether it reads the new or old value."
try to write the same data-structure. If they are not synchronised, the structure may
drop up having some of the value in one writer and some of the other. This is an event
atomicity problem.

The other incarnation of this problem occurs when a process reads a variable and sub-
sequently writes a new value to it. An example: multi-user databases allow multiple
users to access and possibly update their contents concurrently, even when running
on a single processor. If several users are allowed to access shared data, updates may
be lost. Imagine two users wanting to add an amount of NOK 100 to a bank account.
This operation may be split up into reading the current balance, adding the sum, and
writing the new balance back. If each user does this on her own, one after the other,
the result will be that the balance is increased by NOK 200, if they do it in an unsyn-
chronised way and both read the current amount before any of them writes the new
amount back, the account will be updated twice to the same sum, and the balance
only increase by NOK 100, and one update is in fact lost. This is clearly not the
intention. We mention the "database way" of addressing this problem in Section 4.2.
This problem is both an event atomicity, and an event order problem, and has direct
analogies in any parallel system sharing data.

2.1.2 Ordering Dependent Correctness

Correctness will in many systems depend on timing. For instance, programs may not
be able to handle messages out of order. Also, permuted execution may alter the
execution in devious ways. Maples [Map85] gives an interesting example: imagine the
summation of a large array of numbers. Although adding is commutative in general,
this does not always hold on computers. Sometimes one sequence of operations may
give overflow or rounding errors, whereas another sequence may not. Programs should
be written to be robust in all allowed cases, however, in real life this is very difficult
to achieve. This is the typical incarnation of the problem transformations are often
built on a set of assumption, if these assumptions are not valid, the transformation
will not be valid. We simply call this the transformation validity problem.

2.1.3 Deadlocks

Traditional deadlocks are as described in any good operating system or database book,
they are too many to mention and this information can be found in other sources,
too. In short, a deadlock occurs because a participant $\alpha$ waits for another participant
$\beta$, at the same time as $\beta$ waits for $\alpha$. This waiting property is transitive, if $\beta$ waits
for $\gamma$, which again waits for $\alpha$, $\beta$ also in effect waits for $\alpha$.

However, due to limited buffer space and other low-level details, deadlocks may arise
where they should according to the theory not be present. Imagine $\alpha$ waiting for a
message of type $A$. Now, $\delta$ sends such a message, and a proof system not considering
bufferspace would indicate that the deadlock should go away. However, $\alpha$ never gets
this message because the node's incoming buffer its full of messages of type $B$, and the
node will not receive any more messages until some buffer space is freed, by reading
some \( B \) messages. We in effect have a deadlock.

As we see, deadlocks can be a consequence of either unexpected event order or unex-
pected cumulative resource consumption.

### 2.1.4 Partial Faults

A sequential computer either works, or it does not work\(^4\). On a parallel computer,
some of the nodes may function perfectly while others are unavailable. The more nodes
a computer consists of the higher the probability will be that some are “down.” This
introduces extra complexity. It also allows for handling and detecting disfunctioning
parts either in software or hardware, thus opening up for higher total availability. The
subject of fault tolerance is discussed briefly in Section 1.9.1, but is not seen as central
in this thesis.

### 2.2 Performance Bugs

Gene Amdahl [Amd67] explains how the sequential part of a computation must limit
speedup. If the fraction \( n \) of some computation has to be executed sequentially,
speedup is clearly limited to \( 1/n \), even if the parallel parts can be executed in zero
time. This is the so-called Amdahl’s law. In his article, he further argues that the data
management housekeeping part of computation seems to be strictly sequential, take
40\% of the computation, and is fairly constant over time and in different programs.
This claim of his is less than obvious, and has been the topic of some recent discussions.
Some authors claim that in their experience the house-keeping part is constant, and
thus negligible as computation grows\(^5\). Amdahl also gives other examples of factors
limiting oversimplified approaches to high speed parallel computation:

“[..]: boundaries are likely to be irregular; interiors are likely to be inho-
mogeneous; computations required may be dependent on the states of the
variables at each point; propagation rates of different physical effects may
be quite different; the rate of convergence, or convergence at all, may be
strongly dependent on sweeping through the array along different axes on
succeeding passes; etc.”

\(^4\)This is not entirely true, partial service on sequential computers is possible if components other
than the processor fail. In general, services implemented by single components will have simpler
failure characteristics than complex or redundant systems.

\(^5\)The discussion is complicated by some authors arguing about increasing problem-sizes on fixed
numbers of processors, and others considering fixed per-processor problem-sizes on increasing num-
bers of processors. In the former case, relative startup cost decreases, in the latter it stays constant.
One interesting review of this is given by Natvig [Nat91].
According to Rubin et al [RRZ89], the two major sources of inefficiency in parallel programs are

"poor problem decomposition and poor mapping between the program subcomponents and the required resources of computational units."

As we have stated above, many users of parallel computers expect good performance, indeed this may be their very reason for using such hardware. If their program systems do not give them this performance, they will not be satisfied, and probably call it a failure, even though their programs may be functionally correct. The process of getting more out of a system is hence sometimes referred to as "performance debugging".

Gregoretti et al [GMZ86], see three sets of reasons that the processing power of multiple processors is not simply accumulated:

**Contention:**
Participants may compete for resources both on the physical level (memory, I/O), and the logical levels (locks, semaphores, monitors etc).

**Synchronisation:**
Processes have to wait for other processes, e.g. for input.

**Parallelisation of the algorithm:**
We may not always be able to partition the algorithm perfectly.

They argue that these problems can be addressed in two ways, by using a performance prediction methodology and models to achieve early detection of performance bottlenecks, and by programming for observability and correcting the problems when knowledge about execution is at hand.

As Couch [Cou88] points out, swapping and waiting for resources can often be abstracted away as details of little importance in sequential environments. In parallel computers, however, scheduling is essential for results and performance. Imagine two processes, \( \kappa \) and \( \lambda \), on different processors, cooperating and trying to communicate frequently. When \( \kappa \) is scheduled, it sends a message to \( \lambda \), and waits, \( \kappa \) will then probably be scheduled out. When the other processor sees a message for \( \lambda \) it will put it in the ready-queue, but activation may still take some time. By the time \( \kappa \) gets the reply, it has to be reawakened, its pages probably paged in, and so on, all of which takes time. The overall resource utilisation may not be too bad, as other processes are allowed to execute between our two heroes, but the response time is likely to be disastrous. This means that some of the low-level details that we like not to know about in single-processor systems suddenly become very important. This is very unfortunate, because acting according to this knowledge will reduce the portability and generality of the program.

Burkhart et al [BEK+88, BM89, Moe90] have analysed the sources of performance loss, they choose to distinguish between the following:

\(^6\kappa \) and \( \lambda \) are *kappa* and *lambda*, respectively.
2.2. PERFORMANCE BUGS

Idle Processor Loss:
This occurs for instance before a computation is split between the workers and some have to wait for their awakening or when a set of nodes gets one part of work each and the parts are of unequal size, resulting in some having to wait for the delayed partners.

Access Loss:
Due to competition for resources some access control is often involved, and will add some overhead.

Conflict Loss:
Processors will occasionally need to access shared resources at the same time; one will have to wait and suffers from conflict loss.

Garbage-Computation Loss:
This has nothing to do with garbage collection. To avoid idle processor loss the programmer often uses the method of optimistic computation, evaluating result which may not be needed. When these turn out to be superfluous, they are called garbage computations.

Braking Loss:
Some parallel algorithms for finding a valid solution from a set involve letting each participant start with a subset of the possible solutions and see if any of them are valid. As soon as one is found, the problem as a whole is solved. The time from one has a solution until all processors have stopped executing is the braking loss.

Organisational Application Overhead:
Parallel solutions frequently involve giving participants sub-problems, this communication would not be needed if a problem were to be solved by one processor. For instance, splitting an array, sorting the components, and joining the sub-results may be more expensive than sorting sequentially.

Organisational System Overhead:
This is overhead due to the increased complexity of managing resources in a multiprocessor system. If a process is given a name, this may have to be the same for all nodes, i.e. information has to be spread out. The size of this loss is difficult to assess and is by these authors assumed to be zero.

Møller-Nielsen and Staunstrup [MS87] see loss of processing power as resulting from either some hardware phenomenon or software loss, and in their experiments they have come across no significant hardware bottlenecks. Their software losses are broken into: starvation (idle processor) loss, braking loss, separation (organisational application) loss, and saturation (conflict) loss.
2.2.1 Communication Patterns

Athas and Seitz [AS88] see two fundamental approaches to allocating processes with fine-grain concurrency, one is based on knowing connectivity prior to execution, the other is dynamic.

The loss of potential processor power has been addressed by many authors. Wolfstahl suggests [Wol89] that a key to the problem is the mapping of parts of the program to processors.

"Most existing mapping algorithms assume that the communication pattern of a parallel program is fixed and a-priori known. It is shown that this simplifying assumption is rarely true, and that there is a great deal of uncertainty in the communication pattern of a given parallel program."

The uncertainty may take the following forms:

1. The data transfer may vary with time. The communication pattern usually goes through phases, in the extreme case all communication between two participants may take place at the same time.

2. The communication patterns may depend on the input or on the state of other processes.

3. The communication between pairs of participants is rarely symmetric.

4. Spawning and killing processes affects the pattern of communication.

5. Programs may be associated with application-specific mapping constraints (such as the need for floating point units or the "nearness" to specific data). These constraints may also be valid only for a part of the life-time of a process.

2.2.2 Communication and Scheduling Anomalies

In systems where all communication goes through a communication system, the behaviour may vary in a very confusing way. Messages may disappear, be split up, or arrive so late that the system has already decided that something must have gone astray. In some cases, if action is taken to correct erroneous behaviour real confusion may arise if the behaviour was correct but delayed. In the case of independent communication channels, no assumptions can usually be taken about the relative speed of the different channels\(^7\). Such phenomena are often hidden in the lower layers of the

\(^7\)Imagine a node \(a\) sending a message first to node \(b\) and then one to node \(c\). In response to its message, node \(b\) also sends a message to node \(c\). Now, if these messages are not synchronised, node \(c\) may well get the message from node \(b\) before receiving that from node \(a\). This has in some cases been the reason for some confusion in the programming of our own prototype computer HC16 (described below), an error it certainly is, but one which is easy to make and which may often through circumstance stay in the system a long time, surfacing only occasionally.
communication system, or eliminated through the use of a high-level programming model without explicit random messages. In some cases all nodes may function independently, but be unable to communicate due to malfunction in the communication system. If there exists a pair of nodes such that there is no available communication paths between them, the network is partitioned [CP85, Section 7.2].

Bemmerl [BL90] makes the somewhat surprising claim that:

"One of the major reasons for the complicated way of programming existing scalable parallel machines is their inhomogeneous interconnection network. A typical feature of such parallel machines is that the efficiency of the interprocessor communication depends on the localisation of the communication processors within the multiprocessor network."

This results, according to the authors, in parallel algorithms having to be adopted to the network, difficult mapping of programs to nodes, and very difficult and inefficient implementation of dynamic schemes for multitasking and multiprogramming.

Parallel computers can be programmed without considering the topology and characteristics of the communication, and advanced programming techniques will help hide these details. But to get the efficiency of execution that users want, the solutions implemented have to fit well to the underlying architecture. Therefore the parallel programmers have to know and optimise towards machine-specific details.

Durfee et al [DLC87] address the problems of how to choose what to do locally on a node, in order to get closer to a good network solution. Each node in a network operates on some local data, and their solution is to be combined into a network solution. They will also have a choice of what tasks to give priority within the node. Full independence will probably mean redundant work and waste of resources, global control is not assumed to be a good solution since it may create bottlenecks and will make too much depend the controlling node. The article proposes ways of getting coherent computing in a distributed system.

Different issues arise when trying to devise good allocation schemes for sets of processes. For instance, if two processes are both compute-intensive and communicate a lot with each other, how are they to be placed? If they run on the same node, their communication is likely to be cheap, both in terms of cost and latency. At the same time, their only parallelism will be virtual and thanks to time-sharing, i.e. no real parallelism between processors can be exploited. This decision also depends on the nature of communication, in the case of blocking calls execution will be delayed until after reply anyway, and nothing is lost by descheduling the caller and scheduling the callee. More about this in Chapter 4.

---

8 Here, cost concerns the amount of resources "wasted" on communication, like overhead on both sides of a connection and bandwidth over the communication channel. Latency denotes the time from a message is sent until the recipient can act on it.
Chapter 3

Implicit Parallelism

There are several ways of approaching parallelism. Many authors draw a line between implicit and explicit parallelism, and indeed there are very important differences between the two: explicit parallelism is difficult, but fast, implicit parallelism is easy, but in general slow. Admittedly, this statement is so oversimplified that its only effect will be to anger participants from both fields, but there is still a grain of truth in it, as we will argue for below.

We find that this taxonomy is getting a little out of date, and would like to refine it slightly. A new way of thinking arises like an Hegelian synthesis, and lets us split into three classes of parallelism:

Implicit:
In this class we put the languages where parallelism is not a concern of the programmer. This group includes languages where parallelism is inherent in the language and programming model, and the system can exploit it according to well-defined rules without programmer interference. This class also includes the languages where parallelism was not one of the design criteria, but added as an afterthought. As an example of this, take vectorising Fortran. One can deduct from the language what statements can be executed in parallel, but the language itself is not constructed with parallelism in mind.

Explicit:
This class encompasses explicitly parallel languages, and sequential languages augmented with message-passing primitives. The programmer is acutely aware of details of one or more of the issues of location, synchronisation, communication or scheduling.

Casual:
We have chosen this name to indicate that the programmer handles the parallelism in an informal, or casual, manner. An alternative name for this class could be high-level parallelism, as these languages are typically independent of architecture, i.e. on a higher level. This is a class of parallel languages only
recently appearing, with parallelism as a main concern of the language designer. These languages are constructed to support parallel execution and to make it simple for the system to control concurrency, but the user is not allowed explicit control of all details of execution.

It is easy to find archetypal examples from all classes, e.g. vectorised Fortan from the implicit group, Linda from the casual group, and Occam from the explicit group. As with all taxonomies some cases are not so easy to classify because they possess some of the typical properties of several classes.

In this chapter we will discuss the implicit approach to parallelism, in subsequent chapters we will look into the other two classes.

3.1 Debugging with Implicit Parallelism

As suggested in Chapter 2, parallel programming can introduce quite a few problems. When programming with implicit parallelism, most of these problems never appear, since the programmer sees nothing of the parallelism. Synchronising of participants and control of shared resources, for instance, are not of the programmer's concern. A parallel execution of a program has its basis in the same program as a sequential execution, and the result is the same by definition. When debugging a program in an implicitly parallel system, the programmer can restrict herself to debugging a sequential execution\(^1\) of the program, when that works correctly so does any parallel execution. Parallel debugging is trivial in this case, in that it is reduced to sequential debugging. Callahan et al [CCH+88] mention one exception to this without exemplifying:

"Ordinary sequential programming errors might still go undetected when executed on a sequential machine and then appear under parallel execution, but [...] it is fairly easy to design a debugger to help locate errors of this sort."

3.2 Optimising with Implicit Parallelism

Debugging is trivial, but the case of performance debugging is completely different. Since the programmer has no control of allocation of processes to nodes, or indeed no knowledge of multiple processes, finding performance bottlenecks in parallel executions will be a difficult job for experienced programmers with good tools. Removing these bottlenecks will be a question of modifying the source code in such a way that

\(^1\)In other models of programming, we talk about sequential or parallel implementations, in this case the program is the same, only the execution differs, hence the choice of terminology.
the resulting parallel execution is more efficient without altering the result, clearly a process that depends on detailed knowledge, and also the predictability of the system.

The worst part is that to ameliorate, the programmer will have to know all the "dirty details". Since acting on such knowledge only indirectly solves the efficiency problem, later program alterations may undo their effects. The main advantage of programming with implicit parallelism is that such knowledge is not needed, so amelioration will compromise that advantage.

3.3 Automatic Decomposition

S. Gill [Gil58] introduced the concept of time-sharing in 1958, and called it parallel programming. Many of the same ideas and problems as we today encounter in multi-node computers are encountered and commented upon already in this early analysis of parallelism, the author mentions problems of flow charting, keeping control units from anticipating each other, and allocating storage to various participants. The article concerns not only the technical side; according to the author the opinion that parallel programming (time-sharing) would be "impossibly complicated and never worth while" was a popular one in 1958. He argues that it will take time to work out the necessary techniques, and that:

"In fact the advent of parallel programming may do something to revive the pioneering spirit in programming, which seems at the present to be degenerating into a rather dull and routine occupation."

We have since watched time-sharing systems "take over the world," thereby proving the critics wrong. How long will it take to prove critics of today's parallel approaches equally wrong?

In his "Analysis of Programs for Parallel Processing", [Ber66], A. J. Bernstein discusses the conditions\(^2\) for executing code in parallel without altering the result. Blocks of code can be pairwise related in one of three ways:

**sequential**: one block must be executed before the other,

**parallel**: the execution order does not matter, and

**commutative**: the blocks can be executed in any order, but not at the same time\(^4\).

---

\(^2\)We read this anticipation as interfering.

\(^3\)According to [PS85] commonly known as Bernstein's conditions.

\(^4\)This notion of commutative execution seems to have disappeared from the scene of general parallel programming, but still is very important in the field of databases. There, an interleaved execution of transactions is allowed if the result is identical to some serial ordering [BIG87]. In Bernstein's analysis, alteration of execution order is only allowed if all serial orderings of executions give the same result. Interleaving executions of transactions is relevant in databases because a set of blocks together comprise one transaction, this angle is not mentioned by Bernstein.
Bernstein shows that the conditions for deciding if parallel execution of blocks is allowed depend on memory locations read or modified. He also points out that:

".. indirect addressing may prove to be an unsurmountable problem since it may prevent one from determining which memory locations are being dealt with."

This observation is essential for parallel execution of Fortran programs, and one of the key points of the inherent parallelism of functional languages without side-effects. Brinch Hansen [Bri73] agrees when discussing parallel constructs:

"So, unless the programmer is willing to make all [data pointed to by a pointer] private to a single process, [the] pointer concept is inadequate for multiprogramming."

Callahan et al [CCH+88] describe PTOOL, a dependency analysis system for Fortran programs, and ParaScope, a programming environment that will assist in the formulation, implementation and debugging of parallel Fortran programs. The goal of the successor of PTOOL is to combine language specific editing and dependency analysis, to let the programmer interactively see the dependence structure as the program is edited. PTOOL analyses programs as a whole, using *interprocedural dependency analysis*, which is too slow for interactive work. The new approach is to after a modification of a program, check what parts changed, and deduct what parts will need new analysis. Through incremental analysis and improvements, the system will allow much faster re-evaluation of the system, hopefully fast enough for interactive use.

Vectorisation and automated decomposition do not alter semantics of programs, they therefore allow the user to continue to program for the well-known sequential model. Such programs can be debugged on sequential computers, and the entire range of possible errors arising from explicit parallelisation are avoided. The users can also continue to use their "dusty deck" piles of cards from the dark ages of programming, which they seem to like a lot. Since there are vast amounts of Fortran programs already paid for, this implies a major saving.

Reaching a good load balancing and speedup figure is more difficult, there are many specialists making a good living from being able to predict and circumvent decisions taken by vectorising compilers. Their techniques involve manipulation of the way arrays are accessed. Vector processors have special support for repetitive operations on successive elements, and small modifications of code may render the usage of these special functions possible or impossible.

However, improving automatic parallelisation can directly improve all programs written in the language, and thus the saving potential per effort is substantial. Due to the large amount of old programs in use, these people will stay in business for decades to come, before ultimately letting ancient techniques come to a well-deserved rest.
3.4 Hardware

Quite a few projects look into modifying hardware to suit the special needs of functional and logic languages. I have chosen not to look into that area.

Hardware can be parallelised in many ways. Either the user can be allowed to see multiple components, or their presence may be hidden. In the latter case, the programmer may continue to program without considering parallel execution, or she may choose to modify the program to suit the underlying machine better. We have in some cases done this ourselves; in the Intel processors of the various HC computers, successive assembler statements are pipelined, thus exploiting parallelism by interpreting one statement while still executing the previous statement. This pipeline has to be emptied and reinitialised every time a jump is performed. Modifying the assembler code so that jumping is reduced in some cases improves the performance greatly. However, this technique is extremely costly, and will have to be redone after each update of the code. It only makes sense in the inner loops of production quality systems, if at all.

3.5 Functional Programming

When programming in an imperative programming language, the programmer specifies on a low-level what the computer should do. Since this description does not indicate the reason for execution, all statements usually have to be executed. In functional programming languages, the programmer states higher-level properties of a program, and lets the system take more decisions regarding execution details.

Imperative execution fits well with the programmers view of the systems; if a function is invoked in the code, it should be called at the indicated time during execution. Although this may seem trivial, programs need not always behave like we think. Advanced optimising compilers will sometimes recognise that segments of code can never be called, and ignore the code when compiling. In some cases, code can also be restructured to execute more efficiently\(^5\).

Functions may have parameters and return results. The primary effect of a function is that its result becomes known. Sometimes calling a function implies that the state of the program is changed in a way that survives after the termination of the function. We call this a side-effect.

Traditional programming languages allow the programmer to modify variables through memory references (pointers), without knowing where the pointers point to. This makes it very difficult to determine all the effects of the execution of code, or to be more exact, any code using pointers may modify anything at all, and it is very difficult

\(^5\)Debugging optimised code is often not possible because many compilers refuse to lay out debugging information when optimising. There are good reasons for this. Since an optimised program can be changed, the executing code will not correspond to the textual code in the way that we have come to expect. Execution will jump around seemingly at random in the code. This is not good for anyone's view of the world, stick to debugging before optimising!
to determine if any data are definitely not modified. This is the problem of arbitrary side-effects. In this setting, all functions invoked in the code must normally be called, and at the specified time.\footnote{In some extreme cases, the compiler may remove functions not using any pointers or global variables; this is not the normal case in imperative programming languages.}

Global variables are variables which are accessible from all functions, or all local scopes, in a program. They typically lie in the heap. Local, static variables are variables which can only be used inside a function, but keep their value from one invocation to the next.\footnote{Other local variables are usually put on the stack, static variables are put in the heap. As such, they have the scope of local variables and the lifetime of global variables.} Some languages, notably functional ones, do not allow side-effects. If a language has no global variables, the functions are not allowed to have static variables, and there are no pointers, the compiler can predict a lot more about the execution and what it may affect. A function then depends only on its parameters, and hence will return a specific value corresponding to the parameters, irrespective of when or how many times it is called.

Some times the result of a function is never used. It would then be more efficient never to call the function. As we have shown, the effects of a function may be difficult to determine in imperative languages. This is trivial in pure functional languages, a fact which constitutes one of their strong advantages.

Lazy evaluation is described and discussed by Peyton Jones \cite{Jon86}. In short, lazy evaluation is call by need, i.e. an execution model where a function is invoked only when and if its result is used. Compared to traditional execution, this turns everything up-side down. The return statement of a function is examined first, to see what it depends on. Only those statements necessary for finding the return value are executed.

Laziness has several advantages. Knowing how it works, the programmer can program with infinite data-structures and never-terminating loops, laziness will ensure that only the needed results are requested and thus, only the necessary functions executed. Such programs are often clean and clear, as termination is separated from evaluation. However, up to now, it seems that the traditional call by value can be implemented more efficiently. Laziness depends on analysis of program behaviour, this analysis is closely connected to the analysis required for parallel execution. Since functional languages are strong in this field, automatic parallelisation is possible.

It seems that the limiting factor to automatic parallelisation of functional programs is the “fork-out factor”, i.e. how much parallelism the compiler can actually find in a program. If the compiler can only find say five independent activities, adding processors beyond that number will not help much.

At any time, the system will evaluate one expression, and be able to deduct that one or more sub-expressions will have to be evaluated. These can be scheduled at once, since their result will be needed. At the same time, the system will see that some other expressions may be needed. Speculative parallelism, as opposed to conservative parallelism, is to start evaluating some of these expressions. The degree of speculativism
3.6. RELATIONAL PROGRAMMING

can be varied from zero, i.e. conservative parallelism, to infinity, when all expressions are evaluated. Evaluating speculatively is beneficial if the evaluated expressions turn out to be needed later and the extra processing power used would otherwise be wasted. As evaluation gets more speculative, the probability that evaluated expressions will not be needed, increases too.

Another problem is granularity. Independent sections will have to be of a certain size if the cost by coordinating the results is not to overshadow the saving by executing in parallel.

Since functional programs tend to be deterministic and have implicit parallelism, debugging can be done on sequential executions. If a program is correct when executed sequentially, it has to be correct in any parallel execution as well, since such a program is a definition of results rather than a command sequence. The notion of parallel debugging is reduced to performance debugging.

The functional programming language **Lisp** is not lazy, its execution order is as given by the programmer. Halstead [Hal86] shows how to improve parallel performance in Lisp through the use of *futures*, which we describe in Section 4.1.2.1. Futures are a method for manually initiating parallel execution, and are not specific to functional languages. In fact, in purely functional languages future executions can be deduced by the system, and need therefore not be indicated by the programmer.

### 3.6 Relational Programming

Database servers provide services to application programmers. The application program issues calls to the system through a sequential interface. The system has to behave according to a well defined standard, but may well be implemented in a parallel way.

In the server, some or all nodes of a parallel computer solve the query in parallel before the result is returned to the client. The programmer and the users need never know or come to suspect that her sequential view is not accurate, although they may wonder how large amounts of information can be processed at such mind-boggling speeds!

Budd [Bud91] talks about blending relational and imperative programming. In his system, **Leda**, relations, prolog-like inference rules and imperative statements can be freely mixed. The imperative part benefits from backtracking and bi-directional dataflow, the relational part benefits from more direct handling of some issues usually difficult in pure logic programming. This approach should offer very interesting opportunities for parallel execution of the relational parts, unfortunately this project is still young, and the issue of parallelism is not mentioned by the author.
Chapter 4

Explicit Parallelism

In this chapter we will briefly look into different explicitly parallel programming models for parallel computers. The programmer when designing and implementing a system under such models must have a feeling for what the machine "looks like", i.e. how it is programmed, how different parts communicate etc. This programming model depends to greater or lesser extent on several sometimes independent factors where the programming language is one of the most important.

In case of declarative programming languages with implicit parallelism, the only model the programmer has to think of is the base model of the language, be it logic, functional or something else. If the parallelism is explicit, it may still appear on different levels or in different forms. Some paradigms use remote procedure calls, others send and receive messages "arbitrarily". In some models messages must be accepted actively, i.e. a process on the receiving side must explicitly indicate that it wants to receive a message right away. In other paradigms the system automatically invokes some predefined code for responding. Under some programming systems the potential parallelism may be explicit, but the actual location of execution may not be known to the programmer. We will try to sketch how the different models let the user control the parallel aspects of computation.

C. A. R. Hoare tells us in 1978 [Hoa78] that:

"The traditional stored program digital computer has been designed primarily for deterministic execution of a single sequential program. Where the desire for greater speed has led to the introduction of parallelism, every attempt has been made to disguise this fact from the programmer, either by hardware itself or by software. However, developments of processor technology suggest that a multiprocessor machine, constructed from a number of similar self-contained processors (each with its own store), may become more powerful, capacious, reliable, and economical than a machine which is disguised as a monoprocessor."

He also argues that assignment, repetitive constructs (while), alternative constructs
(if), and sequential program composition are the agreed-upon fundamentals of traditional programs. In parallel programming languages, communication in the form of input and output should be the basic primitives.

Browne, Azam and Sobek [BAS89] seem a bit overwhelmed by the intensity and diversity of proposals in this field:

"Each multiprocessor manufacturer either adds architecturespecific parallel-programming primitives to a standard sequential language, proposes a new parallel language, or both. Meanwhile, researchers in universities and industry are proposing many new parallel languages."

They must have had their fill of "paper tigers"; their article ends with this tribute to users and practicality:

"The users have [...] been effective in identifying the weaknesses in the current implementation! In fact, we believe that no one should be allowed to publish an article about their programming environment until it has been used by some threshold number of users."

Perrott and Zarea-Alibadi [PZA86] claim that:

"The development of high-level languages for supercomputers has been ad hoc since their inception. There has been no concerted and well-planned effort to develop languages that carry through the philosophy of high-level programming (i.e. languages based on an abstraction mechanism that frees the programmer from considerations of machine characteristics) and also utilise the parallelism available in supercomputers."

This conclusion may seem surprising. Probably most of the other researchers in the field will claim their efforts well-planned. The statement however specifically refers to supercomputers, in this context defined as vector- and array-processors. It is also our impression that both the supercomputing community and the rest of computing science researchers might benefit from more frequent communication.

Bal et al [BST89] see three issues as central when investigating support for parallel languages:

**Parallelism:**

How is parallelism expressed? Is it implicit or explicit? Is allocation under control of the programmer, automatic, or maybe even dynamic?

**Interprocess Communication and Synchronisation:**

Do processes explicitly send messages, are these synchronous or not? How to communicate with more than one recipient? Who specifies communication? Does the sender know what happened to a message? Can data be shared? Is the system deterministic?
Partial Failures:
Is fault tolerance supported? Can it be implemented? Is it automatic?

Andrews and Schneider [AS83] split into three classes based on their basic mechanisms:

*procedure-oriented* with process interaction based on shared variables. This is also often called the *monitor model*. These languages contain both active and passive objects.

*message-oriented* are based on send and receive only.

*operation-oriented* languages are based on remote procedure calls.

## 4.1 Programming Mechanisms

Before looking into examples, we will discuss some ways of controlling parallelism.

### 4.1.1 Traditional Mechanisms

Brinch Hansen [Bri73] discusses several ways of controlling parallel executions and their effects on programming:

**Fork/Join:**
Brinch Hansen calls these primitives *start* and *complete*. The fork statement splits off one thread of execution which come back together in a join statement. These primitives do not enforce or suggest any particularly structured programming, and are similar to the *goto* statement.

**Concurrent Statement:**
This is suggested by Dijkstra. The concurrent statement says explicitly that a set of sub-statements can be executed in parallel. This is well structured, but depends on the programmer being right in her claim that the statements do not affect each other. If more than one statement updates a variable used by more than one process, confusion will surely arise. Even with indivisible updates, a program may be nondeterministic if more than one of the sub-statements of the concurrent statement update the same variable. If updates are not atomic, the program may also be wrong\(^1\). For the concurrent statement to work we have to control the use of variables and avoid general pointers. This statement is also called *parbegin*.

**Semaphores:**
Semaphores can be used to control the use of variables, or to synchronise processes in general. The operations *signal* and *wait* are provided, and provide

\(^1\)Note that this is completely analogous to updates in databases. See also Section 3.3.
indivisible updates of control structures. These operations are commutative and general, and not timing dependent. However, the problem still remains that the programmer need not make use of these primitives. If one process updates without requesting exclusive access, nothing is gained.

Critical regions:
A critical region is a section of code where we assume mutual exclusion, that the processes inside will terminate in finite time, and fair scheduling. Critical regions can be implemented by pairing the wait and signal calls of semaphores.

Regions:
Regions are critical regions explicitly declaring what shared variables they control. When the compiler has knowledge of the sharing of variables, it can detect illegal access to them.

Conditional Critical Regions:
This is an extension of critical regions, where a given conditional statement is evaluated before entering the region. If this statement evaluates to false, the process is made to wait until it evaluates to true at some later point in time.

Hoare [Hoa74] introduces the monitor concept, where a monitor is an operating system entity, grouping local data together with some procedures to access this data. There is an additional restriction that only one participant can call to the monitor at any one time, other callers are delayed until the monitor is again available:

"[The] bodies of the procedures must be protected against re-entrance by being implemented as critical regions. The textual grouping of critical regions together with the data which they update seems much superior to critical regions scattered through the user program."

The article also mentions conditional entry to monitors and handling of urgency or priority.

Greenbaum [Gre89] defines and describes different level of synchronisation: barrier, neighbour and boundary, and discusses the cost and need of each. Lubachevsky [Lub89] also discusses barriers.

4.1.2 New Mechanisms

4.1.2.1 Futures

Halstead [Hal86] argues the virtues of the "futuristic" approach (use of futures) with respect to parallelising symbolic programs, where one often thought parallelism was not to be found.

Futures are a method for telling the system to prepare a result the program is (probably) going to request later. Instead of calling a function in the normal way, the
4.1. PROGRAMMING MECHANISMS

programmer tells the system to start execution of a function, through a call which then returns right away. The result can be checked by a synchronous call at a later time. This is a general way of increasing the "fork-out" factor, i.e. the number of parallel threads at a time.

In Halstead's paper, futures are used in Multilisp\(^2\). This example is a functional language, but futures are just as applicable under other programming paradigms, Chatterjee [Cha89] mentions their use with C++.

We give a brief example of the use of futures below. First comes the straight C code:

```c
extern int expensive(x);
int result;

result=expensive(x);
```

As we see, the variable `result` is assigned the result of calling the function `expensive`. There is only one thread of execution, and the calling process is blocked until the function returns.

In the next example, the function execution is started asynchronously, and execution carries on in parallel. Some time later the result of the function is assigned to the variable. If the function evaluation is not finished, this assignment will block.

```c
extern int expensive(x);
int result;
future f1;

f1=future(expensive(x));
/* execute merrily along here */
/* ... */
/* some time later : */

result=f1;
```

**Evaluation**  The programmer through the use of a future indicates two things, firstly that a function can be executed in parallel without complications arising, such as the parallel use and updates of variables. She also implicitly says that the parallel execution makes sense, i.e. that there is time to be saved by parallel execution. This latter argument makes futures interesting under various programming paradigms; they can be used as a hint to the compiler or run-time system about how to increase performance. The former argument about data dependencies is not relevant in purely functional languages, because the compiler could have deduced them from data usage,

\[^2\]Multilisp seems to be a side-effect-free Lisp version, with futures added. The question is, does the language have a future?
but the use of futures in imperative languages can introduce parallelism in places where it would not have been detected.

In fact, if the programmer is wrong in her analysis of data usage (or the program is changed) and a future function is in conflict with the main thread of execution, very interesting errors will appear.

4.1.2.2 Shared Memory

Many of the current parallel computers have shared memory, we call these multiprocessors. This memory can be used in many ways, for instance to implement message passing, or as virtual memory, where processes get swapped in on various processors from one shared ready-queue. This shared memory can also be used to share the users data-structures\(^3\).

Some effort has also been put into using the shared memory programming techniques on parallel computers without any real shared memory. One reason for this is that distributed systems scale better than shared-memory multiprocessors [BTK90]. Shared memory can be implemented on distributed memory computers and is then often called Virtual Shared Memory (VSM) or Global Shared Memory (GSM).

This can be controlled by letting one single instance of the memory area live on one node and direct updates that other nodes do to this shared memory to the "residential node". Another possibility is to allow duplicates of memory areas. If this is chosen, the system must state the semantics of their shared memory, and find a way to support them. Ada gives one example of this, we mention the Ada approach in a Section 4.9.

**Evaluation** Advocates of the shared memory approach praise it for its simplicity of programming. Since all data are available, different processors can join efforts to solve a problem without explicit coordination. The problem is that exactly this lack of coordination is very dangerous, it opens up for problems of both the event atomicity and event ordering kind.

4.1.2.3 Virtual Neighbour-Shared Memory

This is a programming model based on some of the experiments with the two first prototype database computers of the Hypercube group: the CROSS8 and HC16-186 (see Section 11). In short it is a way of sharing data-structures between neighbour nodes in a multicomputer, giving the advantages of high-level programming and automated control, while not loosing the performance advantages of allowing for an implementation well suited for the underlying hardware. It is described in more detail in [Moe90c], included here as Appendix C.

\(^3\)Shared memory on Unix uniprocessors is a different concept, which lets several processes use the same memory segment.
4.2. TRANSACTIONS

The paper evaluates the approach, in addition to the description.

4.2 Transactions

Many of the phenomena of parallel computing have direct analogies to problems addressed by the researchers working on distributed databases. They have addressed and formalised the event atomicity problem of parallelism, and use transactions as their atomic unit, see among others [CP85, BHG87]. All relational database transactions should possess the following properties\(^4\) [EN89]:

Atomicity:
A transaction is either fully performed or not at all, i.e. either it finishes or one should observe no results of it.

Consistency:
Transactions should take the database from one consistent state to another.

Isolation
Alterations due to one transaction should not be visible until it commits, in case it has to be aborted.

Durability:
Once a transaction is committed, its results should not be lost due to later failures.

Serializability
The result of running transactions in parallel should be the same as running them after one another in some order.

Unfortunately most parallel programs do not consist of well defined and well separable transactions. However, the transaction concept may offer a good starting point for solving some of the problems of parallel programming, and indeed so it has to Spector et al [SPB88] who made Camelot\(^5\) and Shrivastava et al [SDP91] in making Arjuna. Camelot is a “flexible distributed transaction processing system running on top of Mach. It

“[..] supports the execution of distributed transactions, and the definition, management and use of data servers, which encapsulate shared, recoverable objects. Camelot is based on the client-server model and uses RPC both locally and remotely to provide communication among applicants and servers.”

\(^4\)These are sometimes grouped slightly differently and called the ACID properties.

\(^5\)Camelot is the basis for the commercial product Encina from the company Transarc.
Among other things, Camelot offers: nested transactions, flexible synchronisation, hybrid atomicity,\(^6\) various types of transactions, commit protocols resistant to node or network failures, and logging.

The Arjuna designers use C++ and remote procedure calls to support nested atomic transactions, and provide serialisability, failure atomicity, and permanence of effect. Arjuna objects are *passive*, and reside in permanent object store when not in use. Once taken into use, they are loaded into volatile memory in a server, saving and overwriting is performed when committing a transaction. Objects can be cached and replicated. The application programmer only sees objects, hidden client and server processes take care of the rest, although proper declarations are necessary to invoke the atomic actions.

### 4.2.1 Evaluation

We see that transactions actively address the main underlying sources of bugs in parallel programming: event atomicity and ordering. We will not comment on the ease of programming based on transactions, but it is clear that finding functional bugs is a process that is very well supported by this paradigm.

Since programmers have to decide where objects live and when they are replicated amelioration is possible, it seems that the main criterion for good resource utilisation is the use of well partitioned algorithms.

### 4.3 Guardians

**Argus** [Lis88, LS83] uses a special kind of objects called *guardians*. A guardian is an object which can only be accessed through *handlers*, where all communication is "by value". No direct access into a guardian is possible. A guardian runs at a single node, but can be moved. Handler calls are location independent. A guardian can contain stable objects, these will be reset in case of failure, thus ensuring durability. Argus supports *actions* (short for "atomic transactions") which are serialisable and total, in fact like transactions mentioned above. Recovery in case of crashes are handled by *versions*: when a object is locked, a new version is created. Turning hardware failures into aborts frees the programmer from some considerations, if she handles aborts in general she need not know their reason. If the action commits the new version becomes the current instance of the object, otherwise it is removed.

Argus supports subactions, only one subaction can run at a time, and the parental action is suspended during the execution of the subaction. Subactions are only committed when their *top action* is committed. *Top actions* can be started from other actions, they then proceed independently, thus introducing new parallelism.

---

\(^6\)It is not clear to me what the authors mean by this, but it sounds very database-like?
The language supports the `coenter` statement, where another process can temporarily “take over” a thread of control. A parallel `for` construct allows for parallelism within a `coenter` statement by simultaneously activating several parallel processes. Apart from that, concurrency is allowed only between `top actions`.

Argus is somewhat on the side of the main topic here, as it is designed for systems that are distributed geographically, and that have to behave in an autonomous fashion. They may even be owned by different organisations. It is implemented on top of Unix (according to Dasgupta et al [DJA88]).

### 4.3.1 Evaluation

When the transaction concept is used, Argus inherits their pleasant debugging properties. It seems, however, that transactions in Argus are available, but not enforced. When they are not used, parallel threads of control can interfere with another, and create havoc. Again, load balancing and optimisation is manual.

### 4.4 Distributed Systems

According to Tanenbaum and van Renesse [TvR85] a distributed system is:

> ".. one that looks to its users like an ordinary centralized operating system but runs on multiple independent central processing units."

A distributed operating system is a successor of the network operating system model, with better transparency. Visibility questions occur in three areas: the file system, protection and program execution. In a network operating system, the user will know which computer she is using at any time\(^7\). If she chooses to use several or a particular computer, she has to do so explicitly, typically also identifying herself to each one through some kind of remote login, and also explicitly handle any problems with file location. Distributed systems can be split into three main classes: the “minicomputer model” which is essentially a simple network operating system, the “workstation model”, where each user has a workstation where most work is done, typically on a transparent file-system, and the “processor pool model”, where a pool of processors are dynamically and automatically allocated according to the needs of each user. This can be combined with the workstation model.

Still according to Tanenbaum and van Renesse [Ibidem], the designers of a distributed system are faced with five issues:

\(^7\)A system can be somewhere between a network operating system and a distributed system, in that it may offer full transparency in some fields and less in others. A typical example of this is the Sun Network File System (NFS) where the file system is transparent but execution location is explicit.
Communication primitives:
Distributed systems typically have to communicate using messages. Several of
the other approaches hide the message-passing in some way, like for instance
in the declarative languages and to some extent in the object-oriented model.
Still, they will have to be implemented on some level. Messages can be blocking
or non-blocking. A blocking send returns when the message is sent (unreli-
able blocking) or when it is received and an acknowledge is received (reliable
blocking). Blocking will reduce parallelism between tasks, but eliminate several
potentially unstable timing effects. Message reception can be based on active
or passive reception. Active reception is also called rendezvous, and means that
the receiving process has to actively and voluntarily receive the message, typi-
cally with a receive statement. Passive reception means that messages invoke a
piece of code on the receiving side, set up to handle this case but not explicitly
invoked by the active thread on the receiving side. Message-passing may also
be buffering or non-buffering, allowing some specified or unspecified number of
messages to queue up for reception.

One popular model of communication is the Remote Procedure Call (RPC),
which is very similar to a local procedure call, with the slight distinction that it
involves more than one node and hence more than one thread of control. The
calling thread is blocked until the callee is finished, and the reply to the message
is ready. One problem with the RPC model is that it typically explicitly names
the executing participants, this problem can be overcome by calling ports (See
Section 4.5.1). Another is that it does not fit with any kind of one-to-many
communication scheme.

Naming and protection:
A distributed system may contain several name-spaces. Addresses will for in-
stance typically only be valid within one node, or one job. If names are to be
used from somewhere else, the name-spaces must be coordinated or the system
may use name-servers. These name-spaces may also introduce protection prob-
lems, if the user identification is not unique over all the system. This has lead
to the use of capabilities [MT86], i.e. keys which processes send along to show
that they actually are allowed to perform their requested actions.

Resource management:
In distributed systems, information about the global state of the system is ex-
ensive, and usually not up to date. Knowledge of the resource consumption over
the system as a whole may be inconsistent, some times resulting in unwanted
effects. The resources to be controlled may include CPU-time, I/O-capacity,
buffers/memory, and filesystem. Some systems try to find the parts of the code
that work together, to schedule them at the same time on different nodes. Other
systems try to focus on parts that will communicate in a blocking fashion, to
put them on the same node and reduce interprocess communication cost.

Fault tolerance:
This topic is discussed elsewhere, see Section 1.9.1. Fault tolerance will typically
either be implemented by saving states and tracking messages, or some version of atomic transactions, see Section 4.2.

**Services to provide:**
The designer must choose what services to offer to other nodes, and how to offer them. Among the typical candidates we find file services, print services, process services, terminal services, time services, boot services and gateway services. Some of these are more interesting in a network of computers than within one parallel computer.

**Guide** is a distributed, object-oriented operating system, and then some more. We have chosen to describe this system together with the other object-oriented systems in Section 4.12 instead of here.

### 4.4.1 Amoeba

Tanenbaum et al [TvRvS+90] describe **Amoeba**, and experiences with its use. Amoeba is a system running on a collection of computers on one or more LANs and/or WANs. It provides the user with the impression that she is working on a single timesharing system with rather more power than usual. This transparency is a chief goal of the system. In contrast with network operating systems, the user logs on to Amoeba as a whole when logging on to her workstation. Where and how programs are executed is decided by the system, not the user. The system is designed for distributed computing, but the designers are now looking into using it for parallel processing, where the processor pool cooperates to achieve large speedups on single problems.

Amoeba uses a single, system-wide file system. A file’s position in the directory hierarchy has nothing to do with its location, they may be located in adjacent sectors on a disk or in different countries. File transfer is not necessary, and not relevant, in this system.

Amoeba is based on four components:

**Workstations:**
These are diskless, and primarily used as intelligent terminals.

**Pool processors:**
These constitute the processing resource, one or more of these may be used as the system decides, one user will not know how many she uses, and can potentially use anything between none and all.

**Specialised servers:**
Directory servers, file servers, database servers, boot servers, and others provide services to the users. In some cases services are replicated.

**Gateways:**
The gateways connect Amoeba systems at different sites into a single, uniform system.
The Amoeba kernel provides multithreaded processes, communication services and I/O. Almost all other functions are implemented as user processes. The system is object-based, access to objects is restricted to those with appropriate capabilities, which are cryptographically protected.

Remote operations are implemented as RPCs. The server has to actively ask for a request, and blocks until the client performs a blocking call. The system guarantees that messages are delivered at most once, the client has to handle fault tolerance itself\(^8\).

Each process can consist of multiple threads, which operate in the same address space, but with its own stack and program counter. The thread concept was introduced to increase parallelism in the system. Threads within a process are seen as friendly and are not preempted. The reason for this decision is that critical regions could then be used without locking, since threads had the power to be executing on their own until descheduling. This proved to be difficult, since it is not always clear what system calls might deschedule the active thread, and in new versions threads must explicitly synchronise. In fact, the designers see this lack of preemption as the worst mistake in the design of Amoeba 4.0.

Services are available within their local domain. An object wanting to access a service send a \texttt{locate} broadcast within the local domain. If a server wishes to announce its presence in other domains, it \texttt{publishes} its service, thus generating a \texttt{server agent} in other domains. These server agents will reply to locate broadcasts and forward messages across the gateways. They also initiate \texttt{client agents} on the servers side on the connection. The local user will not notice that she talks to an agent instead of the real service provider, and the server only communicates with the local client agent, which looks exactly like a client. The only parts knowing anything about the remoteness of execution, including its existence, are the agents.

The use of capabilities in Amoeba has worked well. Efficiency of the system as a whole was an important goal, and seems to have been achieved. The use of RPC has worked well, but later systems will be extended to facilitate group communication, to none or all of the participants. Early versions of the system supported asynchronous RPC, where the reply-message interrupted the caller. Although this allowed for considerable parallelism, experience has shown it “impossible to program correctly;”:

"Our advice to future designers is to avoid asynchronous messages like the plague."

\subsection{4.4.1.1 Evaluation}

It may be a bit unfair to evaluate Amoeba as a parallel programming system, since it is primarily a distributed system. We will not be qualified by such considerations! The

\begin{footnote}
\footnotesize
\textsuperscript{8}Reportedly, if a server does not ask for requests, the caller just hangs indefinitely. This will be corrected in later versions by the use of timeouts.
\end{footnote}
existence of threads in one address space is a potential danger to any parallel system, since they require extensive programmer prudence not to give birth to unexpected event orders and event atomicity problems.

That the system allocates pool processors according to need is a nice programming concept, and will probably increase the total system utilisation and efficiency significantly. The perception of a single system removes the danger of overloading the programmer with complex parallelism concepts.

### 4.4.2 LOCUS

LOCUS, described by Walker et al [WPE+83] is a distributed operating system.

"[It] supports transparent access to a network wide filesystem, permits automatic replication of storage, supports transparent distributed process execution, supplies a number of high reliability functions such as nested transactions, [...] partitioned operation of subnets and their dynamic merge is also supported."

LOCUS uses a fully transparent single name hierarchy over it’s participating nodes, meaning that a file has one unique name wherever it is referenced from⁹. Remote execution is invoked through RPC. The programmer executes system calls without knowing if they will invoke some remote service or not, and has no way of controlling the location of execution.

### 4.4.2.1 Evaluation

RPC based systems give a clean programming model, but with low parallelism. LOCUS gives the tools for distributed programming; the use of commits removes the danger of unsynchronised file updates, using it for parallel programming would result in the controlling nodes becoming bottle-necks, and the speedup unsatisfactory.

### 4.5 Parallel Operating Systems

It is difficult to draw the line between distributed and parallel operating system, since many of the systems mentioned below can be used in distributed systems as well. We have put systems not directly qualifying as parallel in the previous section, in

⁹Files can be replicated, all copies are kept up to date by the system. This replication enhances availability in case of failure and performance. When copies of files are being updated, some versions will be out-dated, this can be recognised by comparing version vectors. File alterations are secured using commit, thus avoiding partial changes. Any file belongs to a group, each group has one unique concurrency controlling node. In case of network partitioning, automatic procedures will try to merge according to knowledge about contents of files, if nothing succeeds, the owner will be informed.
this section we present systems displaying parallelism in addition to their possible
distribution. The selection included here is small and incomplete, some are included
because we find them particularly interesting, some because they are mentioned in
the descriptions of current monitoring systems and debuggers below.

According to Douglas, Houser and Taylor [DHE89] multiprocessor operating systems
generally fall into three categories:

- minimal systems not really addressing parallel OS problems, rather like building
  blocks,
- uniprocessor operating systems with simple modifications for multiprocessors, and
- completely new operating systems, typically designed for a specific multiproces-
sor.

Network operating systems are sets of independent operating systems, on communi-
cating computers. Some parallel computers have independent kernels on each node,
and no notion of other nodes. They offer communication as a part of or an extension
of the system, but do not provide any integrated view of the machine as a whole, they
are called microkernels. In this section we describe some of these, and some systems
which give a more integrated view of the computer.

Our operating system, TorOS, falls into this group, but is described later, in Sec-
tion 11.3.

### 4.5.1 Chorus

**Chorus** [Gui82, RAA+90, BCC+91] is a microkernel based operating system. Ports,
tasks and threads have network-wide global names, called *Unique Identifiers*, (UIs).
Chorus implements a minimal microkernel, implementing real-time services. Tradi-
tional operating system functionality is typically provided as a subsystem on top of
this kernel. The following abstractions are provided: Unique Identifiers, actors,
threads, messages, ports (unit of addressing), port groups, and regions (structural unit
in the address-space).

Actors correspond to processes, and are the active units in the system. An actor is
local to a node, and executes sequential code. Actors are awakened by messages, and
execute one *processing step*. After each processing step, control is returned to the
system. When returning control, actors may specify what messages they will accept
next. On reception of a message, code according to the message type is executed within
the reactivated actor. Actors will only be awakened by messages, when specifying what
messages it will accept, timeouts can also be specified. If the timeout is reached, a
special timeout message is sent to the waiting process, thus initiating a new processing
step. An actor is "tied to" one site, there may be several actors per site.
4.5. PARALLEL OPERATING SYSTEMS

Ports can be dynamically grouped, these groups serve as recipients of multicasts or functional addressing. Chorus uses capabilities for access control. Messages are sent to ports, thus abstracting a service from its implementation. A port may only be attached to one actor at any specific time, but may move to a different actor, taking unprocessed messages along. One actor may have several ports for multiple communication paths, and threads may share a port. Port groups can span nodes and actors. RPC and asynchronous messages are available. Messages sent to ports which are not opened, or empty port groups, are simply lost. If an actor dies, another actor may detect this and open ports with the same name as those of the deceased. Messages sent in the meantime will still be lost.

Chorus is designed for a wide variety of computers and networks, and a large class of applications.

4.5.2 Mach

Douglas et al [DHE89] describes Mach (see also [RTY+87]), among other things. Mach is UNIX compatible, the authors see this fact as essential, since Unix in some environments has become a de-facto standard. Among the important functionalities we find: an ability to use large, sparse virtual memories, cheap switching between multiple threads within one address space, flexible memory sharing, fine granularity synchronisation, transparent communication for loosely- and tightly-coupled nodes. The network operating system runs outside the kernel, which is minimal and local. The extensions provide messages, ports, tasks, threads and memory objects. Messages are sent to ports, living on the same node, and all forwarding is handled by the system. The IPC (Inter Process Communication) mechanism uses some advanced copy-on-write memory mapping to ameliorate message passing.

4.5.3 Trollius

Trollius is an open architecture operating system for a concurrent message passing computer, and attached Unix computers [Cen89, Bra88, Bra89a, Bur89, BRDM90]. It is a plain, low-level system, providing message passing primitives, I/O, network information, some kernel operations including process manipulation, and some operations specific to the transputer, like channel and queue operations. It consists of a minimal kernel; some traditional kernel functions are "pushed out" into server processes. The aims of the system include portability and flexibility. Each user can modify a configuration file to customise his configuration of the system. The user has full control over all aspects, including the lower layers, of Trollius, and can for instance remove or replace even components like message handshaking and communication protocols.

In the early period, Trollius was closely connected with the Transputer. We quote from an information leaflet [Cen89] received in 1989:
"As interest in the T-Series\textsuperscript{10} waned, interest in the transputer grew steadily. With no standard operating system for this promising chip, Trollius easily became an alternative for transputer based computers. Today, the goals of the projects are much broader and include other development environments, topology independence, and \textit{any successors to the transputer} [our emphasis].

In March 1991, the Trollius project leader, Greg Burns, sent a message [Bur91] to a mailing list concerning Trollius, informing about the release of version 2.1. This version supported various transputer based systems. Burns also speculate about the further direction of the MIMD world, and writes:

"Part of me is starting to believe that all roads in MIMD lead through Intel. If Intel becomes the IBM of MIMD then we will all take the system software they give us and build on top of it. We are not giving up yet but we are certainly thinking hard about strategic direction, preferably fundable direction."

Trollius can integrate sets of transputer nodes, workstations, supercomputers, and what ever the user may have access to into one LAM (Local Area Multicomputer, see Section 1.3). Burns [Bur89] mentions several advantages arising from this generality:

- Users can access remote computer nodes, i.e. making for instance transputers connected to one workstation accessible to more than one person.
- The compute nodes can take advantage of special devices added to other computers.
- Systems with automatic load balancing systems have more nodes to work over.
- Trollius host nodes can simulate multicomputers, thus easing debugging.
- In the absence of compute nodes, host nodes can be configured as a genuine Trollius multicomputer, thus enabling system development before the intended hardware is available.

Braner [Bra89a] notes that access to a parallel computer is often limited, and advises the programmer to program in such a way that the program can be debugged on the users own workstation. By using conditional compilation (ifdefs) and some additional code, the programmer can prepare the code for running on both a parallel computer and the workstation. The programmer has to do this explicitly; she gets some hints on how to do it, but there is still quite a bit of manual work to be done, and there is no tool to assist her.

\textsuperscript{10}The FFS T-series, an early hypercube based on the transputer.
4.5.4 MMK

Bemmerl and Ludwig introduce the Multiprocessor Multitasking Kernel (MMK) [BL90]. This in an operating system with global objects and operations, meaning that messages are addressed to objects and processes with unique names, and can handle migration. In MMK, the designers tried to:

".. enrich the static language extensions of the conventional approaches with more dynamic constructs and global operations and objects."

Passive objects can be migrated by the operating system, this will probably later be extended to active objects. Their article states that operations on remote objects are based on RPC, and that the communication can be synchronous or asynchronous, buffered and unbuffered, through mailboxes. Synchronisation of tasks can be done by counting semaphores, offered by MMK. Objects are mapped to nodes by editing a configuration text-file before starting the system.

4.5.5 Express

Parasoft provide the Express system [Exp89, Cor88, Te91], which in combination with one of a collection of possible hosts and processing platforms on a cheery note:

"[..] make parallel programming both straightforward and worthwhile. Furthermore Express provides you with the flexibility to move between different types of parallel computers [..]."

Express is essentially based on a per-node operating system with a communication system and some advanced library functions.

Express can use various, different computers at the same time, joining any set of Express participants into a large, heterogeneous “meta-computer.” The user can write explicitly parallel programs with random or structured communication primitives available. She can also convert sequential code to parallel code, thus in fact exploiting the implicit parallelism of the code. It is now (at least) available on various transputer-based computers, NCUBEes, Intel 80386 and i860 systems.

Express tools and debugger are also mentioned in Sections 7.10 and 9.2.3, respectively.

4.5.6 Evaluation

Parallel operating systems give the programmer building blocks for controlling parallelism, but leaves it up to her to use them to construct working systems. The degree of support for debugging depends on how the building blocks are used. Amelioration must be done manually, unless a layer is put on top of the system.
4.6 Data-Parallel Programming

*Data-parallel* algorithms [Hil86, HS86, Wal87, TR88] are called so, because

"...their parallelism comes from simultaneous operations across large sets of data, rather than from multiple threads of control."

This programming is based on a model of a fine-grained parallel computer with general communications, a model more sensible with thousands than tens of nodes [HS86].

Quinn and Hatcher [QH90] tell us that:

"Programming errors occur not so much in the parallel computation implementation with a node as in the interactions between processors. Managing these interactions through low-level asynchronous constructs is error-prone."

They therefore argue for providing the programmer with a more high-level view of the parallel execution, in their view one solution of interest is data-parallel programming.

Hillis [Hil86] describes the Connection Machine (CM), which is central in the work with data-parallel algorithms, since it is an implementation of the machine model on which this programming paradigm is based\(^\text{11}\). Each processor is small, but there are very many of them, typically 16384 or 65536 in the implementations of 1986\(^\text{12}\). The CM extends the memory and instruction set of a host in such a way that certain operations will involve accessing multiple locations in parallel, thus making the CM a SIMD machine. Some instructions are conditional, they depend on a context flag for execution.

In data-parallel computing, memory gets an active role. Processing is assigned to the memory; for example, instead of looping through a set of data-elements to perform some update, each part can typically be assigned to its own processor, and the updates are performed in parallel.

Two characteristics of this programming model are important:

**General pointer-based communication:**

a *send* consists of a receiving address (processor and pointer) and data. The data are put at the address, meaning that a send is in fact a "store indirect".

**Virtual processors:**

Restricting a program to a fixed number of processors would be unfortunate, for portability and generality. Therefore this model gives the programmer an unbounded number of processors, and at execution time maps them to the number of processors available.

---

\(^{11}\)That the CM implements the machine model of data-parallel programs should not be taken to mean that all CM programs are data-parallel.

\(^{12}\)The Connection Machine has come in several versions after this.
The authors give some interesting examples of data-parallel programs, among others:

- array summation, where each element is active and summation takes place in a logarithmic number of steps,
- list-processing, where each element of the list has its own processor processing the pointer to the next element,
- document retrieval, where many documents are searched in parallel, one processor per document
- VLSI design, each cell, cell location and connection point represented by a processor
- circuit simulation, each component in a processor,

The authors have experienced that data-parallel programming is more general than they at first thought, but requires a new way of thinking. They also observe that:

"[..] if the number of lines of code is fixed and the amount of data is allowed to grow arbitrarily, the the ratio of code to data will necessarily approach zero. The parallelism to be gained by concurrently operating on multiple data elements will therefore be greater than the parallelism to be gained by concurrently executing lines of code."

Quinn and Hatcher [QH90] summarise the benefits to be collected from this approach: control flow is simple, computation results are deterministic and independent of the number of physical processors, building a debugger is straightforward\(^\text{13}\), and data-parallel programs are portable.

Still they repeatedly insist that data-parallelism is only one solution; it is good for quite a few things, but unsuited for others:

"However, a data-parallel language is inappropriate for implementing programs with multiple asynchronous processes, like database management systems and multiprogrammed operating systems\(^\text{14}\)."

### 4.6.1 Evaluation

Breakpoints make sense in data-parallel programs, enabling that popular method of debugging. They are also well suited for event-based debugging, if the sequence of communication is traced, a replay of one node with regenerated messages would

\(^{13}\text{Debugging is here straightforward because it can be done just like sequential debugging; we have termed this trivial.}\)

\(^{14}\text{We are a bit surprised at finding an OS in this list, as we would rather see that as a method than a goal.}\)
be enough to see everything going on in that node, and timing is hidden from the programmer.

Due to the synchronicity it should be easy to visualise the level of parallelism and the resource utilization over time, which is a good basis for improving them.

### 4.7 MIMD Data Parallelism

If data-parallel computing lets the programmer manage the complex task of programming a parallel computer, it is definitely a good thing, and one which should be pursued. Quinn and Hatcher [Ibidem] discuss how data-parallel programs can be executed on MIMD computers as well as on the more obvious hardware, SIMD computers. They tell us that many MIMD programs fit into the SCMD\(^\text{15}\) style (Same Code, Multiple Data) which is close to data-parallelism, anyway.

SCMD programs run the same code on all processors, executing on local data. Typically these programs operate a while on the local data, then communicate to partition data or (intermediate) results, implicitly synchronising when communicating. This is repeated until termination. This can be done without explicit message passing in data-parallel languages like C\(^*\), an extension of C with data-parallelism primitives. If the programmers formulate their solutions in C\(^*\), these programs could easily be converted to C, and executed on multicomputers. This conversion can be automated, and is thus cheap\(^\text{16}\).

#### 4.7.1 Evaluation

Debugging data-parallelism is trivial, not because it can be debugged as a sequential program like with implicit parallelism, but because it has synchronous threads of control. MIMD data-parallelism would keep enough of that property to allow for inspection before and after synchronous communication, enabling a well defined, step-wise debugging model.

Performance visualisation can be based on the communication synchronicity.

### 4.8 Communicating Sequential Processes

Hoare [Hoa78, LS84] makes an ambitious attempt to find a solution to all the problems of parallel computing, which consists of a language with the following essential features:

\(^{15}\)Also called "Same Program, Multiple Data".

\(^{16}\)In this age of computing, virtually anything which can be solved without human interference is cheap, and Yes, the context of this claim is computing!
4.8. COMMUNICATING SEQUENTIAL PROCESSES

- Guarded commands, which control sequential execution and give the only source of nondeterminism.

- The \texttt{parbegin} statement controls and specifies parallelism (see Section 4.1.1). Here, a set of statements are allowed to start execution at the same time and execute in parallel. The \texttt{parbegin} statement itself finishes when all its components finish.

- Simple communication primitives are introduced, consisting of \texttt{process!variable} for sending, and \texttt{process?variable} for receiving.

- Communication primitives specify both sender and receiver. These primitives are synchronous, and occur without buffering when both the sender and receiver reach their respective statements and name each other.

- Input commands may be guarded, i.e. they are only executed if there is someone wanting to input.

- Repetitive statements may have guards.

- A simple pattern-matching feature gives the programmer a way of not getting messages not matching the pattern, and also to securely access parts of messages.

This language is rather static, it may be executed with less processors than it is programmed for, but processors past this limit will not be used. The naming of senders and receivers increases determinism, but makes the language less general, less useful for libraries, and severely reduces its re-use potential. It is not constructed for automatic verification or proofs. Hoare is determined not to specify fairness into the language, since that increases the complexity of the implementation and possibly confuses the semantics; in a non-deterministic language, the programmer will have to take care of termination anyway.

CSP has strongly influenced the language \texttt{occam}, which is implemented on several platforms and in common use.

4.8.1 Evaluation

The fact that communication is synchronous and uninterrupted implies that the debugger can halt and restart processes for inspecting their data without changing the functional behaviour of a program, therefore one process can be debugged on its own. This is very useful. The event atomicity and event ordering problems are well handled under this model.
4.9 Ada

The language Ada [Uni83] is developed by the US Department of Defense. It is meant to support embedded systems, therefore can handle exceptions on several levels and real-time execution. It seems very complete, and is extremely large; some critics complain that all ideas from the past forty years of programming are put into it. Its followers think this is a good thing, anyone can do everything in Ada. We will not here concern ourselves with the plethora of possibilities in the language, but rather look at the way processes interact.

Ada uses the rendezvous, a protocol where two consenting processes can choose to communicate. One of the participants offers its service, without specifying to whom, it executes an accept statement. This offer may have a timeout, and there is a way of specifying several services at once, where any one but only one offer can be taken. The other participant has to specify exactly what services it desires, and from whom. This looks exactly like a call and may give parameters in, and receive values out. There is one thread of control on each side of this connection, the naming participant is held up until explicitly released from the serving process. After releasing, both processes keep on executing.

This protocol is fairly general, one obvious way of using it is to have tasks “spinning” over an accept, to act as a server and reply to anyone wanting something until told to quit. Another is to let processes accept and call according to a predefined pattern which then all tasks have to know.

Ada also offers shared variables, the update semantics are a bit unusual. We see in [Uni83, Section 9.11] that synchronisation takes place at the start and end of rendezvous between tasks, and that tasks reading or updating shared variables between two synchronisation points may assume that no other tasks update or read during that interval. Bal et al [BT88] call this semantic weird, and we have to agree that it is quite unusual. If a programmer does not observe this updating sequence, i.e. if she lets different tasks read or update a variable without synchronising, the behaviour is not defined and is likely to be the source of some fascinating bugs.

4.9.1 Evaluation

Ada is a very large language, and offers the programmer a wide variety of opportunities. If shared variables are used, event atomicity and ordering may give problems. Apart from that, parallelism is integrated in the language with extensive use of the strong typing mechanisms and avoidance of pitfalls like unsynchronised sharing. This should open up for trouble-free parallelism.

Again, load balancing and amelioration are up to the programmer.
4.10 Cantor

Athas and Seitz [AS88] describe the Cantor programming environment, which is a "programming notation for writing message-driven programs using concurrent objects." Cantor is strictly reactive, execution takes place only as a response to a message. Objects have private variables, and a sequence of actions describing how the object will respond to messages. Objects must process messages in a bounded number of steps, and sleep until the reception of new messages. References to yet unbuilt objects are called futures, their use can increase load balancing. Concurrency is introduced only when an object sends more messages than it receives. According to Seitz [Sei89] reactive, message-passing programs are for most purposes, equivalent to Actor computations (See Agha [Agh90]).

Seitz and Athas [Sei89, AS88] advocate the "Sweep-model", which is a way of modelling parallel computations. They model programs behaving like Cantor, where messages sent to a node result in one action being taken on the node, possibly generating new activity. One sweep in this model is a synchronised step where all nodes process one message. Sweeps are modelled as taking constant time, so is the time between sweeps. Under this model, one can count the number of participating objects at each sweep, this gives the maximum amount of parallelism in the program at that time. Seitz claims that this modelling is useful, because its behaviour resembles the reality.

4.10.1 Evaluation

Object-local variables seem to be protected, and if we assume that any object can only process one message at a time, event atomicity and ordering are well handled. If the pure reactivity implies that any action can be taken at any time, given the right message, each action must be valid at all times. This seems require the writing of very general code, and create a danger for running into unexpected situations. Again, load balancing is manual.

4.11 Problem-Domain Structuring

Fox et al [FJL+88] describe the CrOS III operating system. The communication system is in fact "just a collection of routines which can be called by application programs." In addition to the traditional send and receives, Fox introduces something he calls expected communication, also called loosely synchronous communication.

The main idea is that all nodes call the same communication routine at the same time. This is used for communication where all nodes have one piece of information, and this information is to be redistributed in some way. It is loosely synchronous, because the nodes call the routine at leisure, but no node can continue until the information it needs is available.
As a simple example, let us take the function \textit{vm\_combine}. One of the arguments to this routine is a function, another is a buffer. When called, the system executes the function over all the data supplied by the nodes, and returns it to all the nodes. If a program needed to determine the total number of say records over all the nodes, the call:

\[
\text{all\_records} = \text{vm\_combine( my\_records, add\_func, sizeof(int), NODES):}
\]

would give the result. In this example, \textit{my\_records} is the local result, \textit{all\_records} is the global sum, and \textit{add\_func} is a function which adds over a vector.

Fox suggests using vectors instead of single numbers as arguments to such calls, and gives a list of calls operating over them.

4.11.1 Evaluation

How these calls are implemented is of little interest to the programmer, and thus enables the system to optimise. Such a system can check that all nodes call the correct functions, and that the parameters make sense, thus removing the problems of event ordering due to random messages.

By tracing local arguments to and result from the collective calls, each node can be debugged on its own. Nodes not adhering to the communication structure can be trapped by checks in the run-time system. As we see, a system of this kind is very debuggable.

Also, this approach gives a very natural load-balancing, since the programmer from the start on thinks in terms of distributed structures.

The one thing that can limit the usefulness of this approach is if an application refuses partitioning into subproblems. Fox argues that most problems fit into such a structure, and gives many examples to support his claim.

4.12 The Object-Oriented Approach

As one of the terms currently most commonly used within computing, object-orientation (OO) of course offers a new and interesting angle to parallel programming. There are many efforts within this field, and we will comment upon some of the typical properties.

4.12.1 Orca

Bal and Tanenbaum [BT88] discuss the \textit{shared data-object} model. In this model, objects are shared, and can only be accessed through indivisible operations. By letting
the granularity of such operations be large enough, the programmer can effectively remove the problem of asynchronous updates to single objects. The implementation distinguishes between reading and updating operations, and keeps track of the access pattern of processes to objects. These statistics are used for selective replication and migration of objects; the introduction of multiple replicas of an object means that the system has to keep track of and update the various versions. The article gives an in-depth discussion of how to do this, without letting the programmer know. Bal et al [BTK90] describe the programming language Ora, based on the shared data-object model.

### 4.12.2 Eden

Lazowska et al [LLA+81] describe Eden, which is an early attempt to integrate a computing environment. In Eden, each object has a unique, system-wide name, and can be accessed if the requestor has the right set of capabilities. Details of location, concurrency and error recovery are encapsulated within the object. The user sees simple objects, her thread of control is suspended when she calls the object, i.e. it looks like a procedure call. The typical use of Eden is on workstations, where a workstation behaves like one, but at the same time is part of a larger pool of resources which other users can draw from. Eden is based on the microkernel view, where a traditional OS is built on top of the building blocks in the kernel. Eden objects have active processes, which may die. Active objects are backed up to passive objects on backing store, but the user operates with single-level objects. In the case of an access to an object in the passive state, the object is "reincarnated" before it can respond. This is somewhat analogous to swapping in a process in traditional operating systems, but not quite since the coordinator process is recreated, not merely swapped in.

In Eden, locality is relative to "nodes", which in this context supplies virtual processors and virtual memory. An active object is at any time supported by one node, but one physical node may contain many virtual nodes, and a virtual node may reside on a set of physical nodes.

### 4.12.3 POOL

The Parallel Object-Oriented Language (POOL) [Ame89, AR89, Bro89] provides the user with virtual nodes. When an object is created, it is placed on one such node as specified by the user. The number of virtual nodes is infinite and independent of the number of real nodes available, currently virtual nodes are allocated to real nodes in a round-robin fashion. Virtual nodes never move about on the real nodes, and objects normally stay on the virtual node where they were created.\(^\text{17}\) The programmer should place objects that communicate a lot on the same virtual node. The system takes care of the rest, this implies that the programmer can ameliorate the internal vs. external

---

\(^{17}\)Objects can be copied and destroyed, thus in effect moving the object.
communication, but not utilise any information about the nearness of other nodes, probably a good thing for architecture-independent applications.

### 4.12.4 Guide

**Guide** [BDF+87, RS91, BCC+91] is an object-oriented distributed operating system. In Guide, a job is a multi-threaded\(^\text{18}\) virtual machine which may dynamically and without the users knowledge diffuse to several actual nodes. A job may exist on several nodes, and a node may be visited by several jobs. Objects may be dynamically bound to jobs, and may be shared between jobs. The main foci of the project are on location-transparency and high-level services.

"[...] the user interface hides most of the problems of distributed resource allocation, thus making the network appear to behave like a single large time-sharing system."

The system supports persistent objects, keeping information in objects instead of traditional files. It has been implemented on top of Chorus, Mach, and Unix.

### 4.12.5 Clouds

Dasgupta et al [DJA88] describe version 2 of **Clouds**. This is an operating system designed to run on a set of general purpose computers, uni- or multi-processors. It is not clear if these computers need to be homogeneous. Clouds uses a minimal kernel, called **Ra**, which supports the basic function of the system: location independent object invocation.

An object in Clouds is passive, and not bound to any process or server. It is a virtual address space and also permanent, this is significantly different from other systems. Objects have unique names, and consists at least of a code segment, a data segment, and a memory allocation scheme. The data segment is not accessible outside the object. Objects have *entry points*, each with its own *operation*. This object model merges the idea of internal memory with structure in computers and files on backing store.

The active entities in Clouds are called **threads**. Threads can invoke objects, this involves entering through an entry point and executing from the objects code segment, with access only to the objects' data segments and the parameters of the active invocation. Parameters are transferred in and out by value only. There are no "pointers" across object boundaries. Several threads can enter and execute simultaneously. Threads can span objects and machine boundaries. Machine boundaries are not detectable in the system.

\(^{18}\)Threads are called *activities* in Guide.
Control transfers between address spaces occurs through object invocation, data transfer through parameters to and returned values from invocations.

The user’s view of the computing environment is simple; it consists of named persistent objects. There is no need for backing store or messages, so this is not provided.

Threads going astray would create havoc, if allowed to do so. To avoid this, Clouds introduces atomicity. All operations on objects are tagged with a consistency label, showing how important it is that the operation is performed in an atomic way. A thread assumes the level of atomicity given by the operation when entering. Updates are performed on copies of the objects data. Updates performed by standard threads are written back at leisure, consistency preserving threads use two-phase commits when the thread finishes.

### 4.12.6 Evaluation

Strong "walls" around data structures is a good start in parallel programming, and in this respect OO scores very well. That internal data structures can only be modified by invoking an objects methods, removes the problem of inconsistent updates. The problem can however be reintroduced by letting the objects offer operations of too limited functionality. If an object offers the two independent methods read and write, the "lost update" problem is back.

Under this model, communication is specified in terms of objects. The object location should be of less importance, since communication between two objects has the same form independent of their relative location. The encapsulation of OO helps the programmer handle parallelism.

Independent execution of objects is an integrated part of many OO systems. Communication and concurrent access is thus a central part of such languages. Debugging parallel OO programs should ideally be trivial, in our usual sense that parallelism makes no difference. One can also argue, however, that parallelism between objects introduces parallel problems also within one node, and that location independence at that price is rather expensive.

However, in an OO system with a large number of active objects, i.e. objects with independent threads, the question remains how to efficiently exploit the power of a parallel computer. In short, what objects go where and who is to decide?

### 4.13 Delayed Mapping

In this section we describe two approaches where the specification of the code to be executed is divided from the specification of where it is to be executed. Note the similarity between this and bilingual programming, as mentioned in Section 5.4.
4.13.1 CODE

The Computation-Oriented Display Environment (CODE) is described by Brown et al [BAS89]. Their goal is to describe parallel computations without implicitly programming for some specified architecture. They achieve this by separating a parallel program into two basic elements:

**Computation units**: describing the functionality of the code, and *firing rules*, i.e. a set of guards over the input-dependency set specifying when a unit can be executed.

**Dependency relations**: which compose computation units into a structure of parallel computation based on data, demand, mutual exclusion and control dependencies.

Programming is done by filling in forms for nodes and arcs in a generalised dependency graph. After specifying all details required, a CODE tool generates a program independent of architecture. This program is then mapped to some actual execution environment through the use of another tool, and since this conversion is automatic, porting a program is trivial. It is not quite clear to us why the computation units and dependency relations should be separate, if they together constitute a program specification independent of architecture. This may have to do with reusability of code.

4.13.1.1 Evaluation

It seems that the inputs of a computation unit could be traced for re-execution and debugging purposes, since they are independent. Their separation must also mean that event atomicity and ordering are removed as problems.

Automated reoptimisation for new architectures is very elegant and efficient, if it works. We are not sure how well developed these processes are up to now.

4.13.2 IMPACT

Burkhart et al [BEK+88] describe IMPACT, which is part of the project $M^3$. In this system, parallel processes are explicitly declared, each consisting of sequential code. Program development takes place in two phases: programming and configuration. In the programming phase the programmer formulates the sequential parts of the program, using a variety of process synchronisation and data-access primitives. The algorithm is split into cooperating processes, in the form of a process graph. IMPACT will automatically generate code skeletons for the programmer. In the programming phase, the programmer finishes the program as such.
The next phase is the configuration phase. Here the programmer defines groups of closely connected processes into tasks. Closely connected processors are grouped into pools. The configuration of the system consists of assigning tasks to pools; this is the only architecture-dependent part of the process.

### 4.13.2.1 Evaluation

Compared to CODE (above) it seems a pity that the programmer must participate in the architecture dependent phase. Since we do not know what speedup numbers the different systems have, it is difficult to argue about whether the extra effort is worth while.

### 4.13.3 Mapping Directives

In Section 2.2.1 we mentioned the argument that communication patterns may change over time, as argued by Wolfstahl [Wol89]. Given a system where processes may be moved from one processor to another, rescheduling might be automatic, responding to the observed patterns. This would require tight observation and quick rescheduling, and would be prone to late responses. In the case where patterns vary rapidly, relocation might actually decrease performance by always adopting to patterns of the past and being unprepared for the present. [Ibidem] goes on to suggest a solution. Processes must be augmented by mapping directives so that the knowledge of the programmer about foreseeable effects on the communication can be taken into account before they actually happen. Then, remapping can take place in time to be effective when needed.

#### 4.13.3.1 Evaluation

With explicit parallelism, the programmer decides where the processing takes place. With casual parallelism this is usually not the case. The mapping directives give the programmer a way of hinting to the system how she thinks processing should be distributed. It is very similar to using the "register" storage class in C\textsuperscript{19}, it may not help much, but it can hardly do any harm.

Mapping directive should have no implications for debugging.

We assume, however, that the directives will only to a limited extent be heeded, and the program development will typically lead to a directive being acted upon only now and then. This may be a frustrating experience for the programmer.

If execution location is really important, the programmer should consider using a more

\textsuperscript{19}The "register" storage class in C lets the programmer tell the compiler that she thinks it would be a good idea to keep some variable in a register, usually because it is used a lot, for instance in a tight loop.
explicit paradigm. Mapping directives may however be useful for parallel programs where performance is not of utmost performance, but where the programmer feels good about sharing her insight in a cheap and uncommitting way.
Chapter 5

Casual Parallelism

As we have seen, implicit and explicit parallelism both have their problems, but also advantages. As time goes by, a synthesis of the field arises, in our case this is represented by what we call casual parallelism. In this class, parallelism is a central issue, not added as an afterthought. The parallelism in the programs is specified on a high level, to achieve independence of architecture.

One of the main problems being addressed is non-determinism. In some systems non-determinism is removed, Strand is an example of this. In other systems it is allowed to stay, but redefined to be a feature instead of a flaw; Linda is a good example in this case, since its very nondeterminism is what opens up for flexible scheduling.

All casually parallel systems enforce indivisible updates of variables. The degree to which these systems remove the programmers possibility of making a mess through inconsistent and interleaved updates of datastructures with relationships varies. In most systems the programmer still has a chance of making logical errors resulting in deadlocks and related problems, but in some cases this now requires some ingenuity.

In good casually parallel systems, the user program is a description without low-level details. This increases the chance that such programs can be moved between and behave decently on various hardware platforms.

5.1 Debugging and Optimising

The abstraction level also makes it possible to debug on a higher level. We will discuss this along with the description of the various approaches.

As part of the price paid for the high abstraction level, we have to live with a lack of detailed control. Indeed, that is why it qualifies as “high level”: the details are hidden. This also implies that detailed tuning by the programmer is not possible, and should not be possible. However, a high-level language also lets the compiler analyse how and when data are used, and thus enables advanced optimising compilation. This has the
advantage that the programmer has less specification to do and will probably make fewer mistakes. It also means that the compilation system can generate different code appropriate for the characteristics of different parallel computers. Another advantage is that the application can benefit from advances in compilation technology for free.

5.2 Bucket Programming

In this section, we have collected some approaches based on independent agents having nothing in common but a bucket where “things” are deposited and retrieved. These buckets are given various names, Linda uses a Tuple Space, also called a “bag of processes”, heaps is also used as a name for this.

5.2.1 Linda

Linda has a mission [ACG86]:

".. to make it largely unnecessary to think about the coupling between parallel processes."

As [Ibidem] also points out, Linda is not in herself a programming language:

“When the simple operators Linda provides are injected into a host language h, they turn h into a parallel programming language.”

The basic unit of all Linda programming is the tuple. Linda’s processes are uncoupled, and never directly deal with each other. Instead they indivisibly place and withdraw tuples, from a tuple-space by commands called in and out. A communication between two processes takes place in two steps. First the sending process places a tuple into the tuple space. Then some other process issues a request for a tuple. If a matching tuple is found, it is returned. If several tuples match, just one is returned.

If two processes post identical requests and only one tuple matches, just one of the processes will get a tuple; Linda is clearly nondeterministic.

In addition, the tuple space offers the primitives read, to see a value in a tuple without removing it, and eval, to place an unevaluated tuple into the tuple space. Requests can specify partial matches for a tuple, by putting values in some of the fields of the tuple request.

A parallel program is a spatially and temporally unordered bag of processes, instead of the common process graph. The authors talk of processes as process tuples, a program is started by dropping one such process tuple into the tuple space, letting it drop other tuples in.
5.2. BUCKET PROGRAMMING

Gelernter [Gel85] gives an early overview of "generative communication" and the ideas behind Linda. The distinguishing properties of Linda are:

**Space uncoupling:**
A tuple is referred to only by its contents, not its sender or location.

**Time uncoupling:**
Tuple put into the tuple space are there until they are removed. Processes trying to get tuples out of the tuple space are blocked until they can get something out. Timing matters for nondeterminism, but it is not a part of the programming.

**Distributed sharing:**
Since communication goes through the tuple-space, processes with disjoint address spaces can share a variable by placing it there.

**Support for continuation passing:**
By this the authors refer to:

"[..] one process's sending data to another, blocking in expectation of a reply, and being continued ultimately by a third (or some nth) process."

**Structured naming:**
Specific communication can be implemented by putting identifiers into the tuples, and using these as parameters to in.

The problems stemming from Linda are two, tuple space does not implement security, and tuple space may be difficult to implement well.

Carreiro and Gelernter [CG88] argue that Linda's usefulness can be defended with three arguments:

- Linda is being used to solve "real" problems,
- the Linda solutions to these problems are easy to understand, and
- they demonstrate real speedup.

### 5.2.2 Problem Heaps

Møller-Nielsen and Staunstrup [MS87] note that vector and pipeline algorithms are static with respect to processing elements, i.e. they have a fixed upper bound of parallelism. This is unfortunate, since it limits generality and portability. When an algorithm does not exploit the potential of the computer, i.e. when the efficiency (see Section 1.5) is less than one, this is due to software loss (Section 2.2). Their method consists of splitting the task into a set of subtasks which can be executed in parallel. Subproblems are put into a problem heap, which is shared between the processors. A
set of identical and asynchronous processes take problems from the heap, and try to
solve them. This may result in more problems being put on the heap. All processes
run the same program, which looks more or less like the example in Figure 5.1.

REPEAT
   take a problem from the heap
   IF the problem is simple
       THEN
           solve the problem
           include the result in the solution
       ELSE
           split the problem into subproblems
           put the new problems on the heap
       ENDIF
UNTIL heap empty and all processors idle

Figure 5.1: The main program with problem heaps

This approach has several advantages. Processes are identical, and any number of
them may be used, the parallelism is implicit. The parallelism in the execution is re-
stricted by to which degree problems are split into subproblems, but not by anything
else. An important issue in getting efficiency is the granularity; the ideal granular-
ity unfortunately seems to depend on the efficiency of problem heap accesses, and
thus on the relative speed of the communication. If the granularity is too fine, the
implementation will suffer from organisation application overhead and conflict loss,
if granularity is too coarse, braking loss, idle processor loss and lost parallelism will
result.

The structure of the problem heap must suit the application; stacks, sorted lists and
heaps may all be adequate for different problems.

5.2.3 Holistic Algorithms

Holistic algorithms are described by McKeown [McK80], and used in work described
by Storey [Sto89]. Holistic algorithms are said to be:

"... a MIMD-paradigm for defining the highest level of parallelism possible
in the design of an algorithm."

Holistic algorithms are based on an "archetypal task", which must contain the whole
solution to a problem and be able to run alone or replicated. This task must be
iterative, and reach a solution quicker as replication increases, at least up to some
non-trivial limit. It works on a global data set, whose contents may be modified but
where the form is the same. Each processor runs exactly one task.
The only significant function of the operating system is to ensure that global data are accessed properly. McKeown suggests using the critical regions for concurrency control, but Storkey suggests that the monitor concept is better suited. Both mechanisms are described in Section 4.1.1.

It is not clear to us why each processor could not execute several instances of a task. Implementing several general logical nodes per physical node in this way is a well-known and general method, and would improve performance due to the normal time-sharing arguments.

5.2.4 Comparison

Linda, problem heaps, and holistic programming are clearly related, but Linda is a set of language constructs, and the two other programming methodologies, relatively independent of language. In all of these systems, communication takes place only through the bucket. The operations valid on the bucket vary somewhat; in the case of Linda they are well defined and general, in the other cases they are just suggested, but indivisible operations are essential to all approaches.

In Linda, processes live in tuple-space, just as data; holistic programming explicitly uses one process per node without relating it to tuple-space. Also, Linda allows for different tasks, both the other approaches assume that all tasks are identical.

Linda tuples have types, widely varying forms, and can even be active. This is clearly more general than the two other cases. With problem heaps, problems are put on the heap, whereas holistic algorithms focus on data. This suggests a difference; if we draw an axis from having processes in tuple space in Linda to pure, isomorph data in holistic algorithms, problem heaps are somewhere in between.

5.2.5 Evaluation

Debugging bucket programs should be easy. All communication between processes goes through the bucket, therefore each process depends only on what is gets back from its atomic communication with the bucket, and possibly its own previous state. The complicating factor that the local computation of one process depends on the state of some other participant is not present.

If the individual processes are identical and state-less (or purely functional) as well, the setup is perfect. Identical processes reduces the number of special cases the bugcatcher has to think about. That each invocation is state-less, i.e. that it depends solely on its input, implies that even the order of tuples handled by one process is irrelevant. Then an executing system need only save a trace of all the operations on the bucket, and this provides all information needed for debugging each invocation individually. Different processes asking for overlapping sets of tuples will introduce some difficulties, then timing will be critical for which process gets a tuple, making it
non-deterministic. Also, surviving local state in a process also removes determinism, as the order and identity of processes getting tuples will then matter.

The beauty of this approach is that performance debugging should be superfluous due to the automated scheduling and allocation, the ugly side is that this may lead to “stupid” usage. Imagine a program running on a computer with ten processors. The program consists of ten tasks called \( n \), one task called \( m \), and one thousand tasks called \( t \). The cost of \( n \) and \( m \) are 100 time units each, an instance of \( t \) takes one time unit to finish. The problem is solved when all tasks are completed, the only other dependency is between \( m \) and all instances of \( t \): all \( t \) require that \( m \) is finished. In a case of bad scheduling, all \( n \) would be scheduled first, at time 100 they are all finished, and only \( m \) can run. Nine processors do nothing until \( m \) is finished, at that time instances of \( t \) are enabled, and will finish in one hundred time units. The total cost of this is three hundred time units. In a case of good scheduling, \( m \) and nine instances of \( n \) are scheduled at first, after one hundred time steps one processor will take on the last \( n \) whereas all the others start on the \( t \)s. This is an optimal allocation, and takes only two hundred and ten time units. Often the programmer has knowledge about the application which could improve performance dramatically as often done in explicitly parallel programming. In many cases, not requiring this knowledge leads to cheaper development of more robust and durable programs\(^1\).

5.3 Strand-88

Strand is a logic programming language. Strand-88 began as a language, but soon grew into being a programming system. It is based on an inherently parallel model of execution. The language handles dynamic load-balancing. Execution is guaranteed to execute consistently across all supported hardware platforms, by data-flow synchronisation of processes. This must mean that the language is actually deterministic, which opens up for lots of nice things!

Foster, Overbeek and Taylor [FO90, FT90] describe Strand as a member of the family of “concurrent logic programming languages.” It has extremely light-weight processes, user-defined types and automatic storage management. Processes communicate by reading and writing variables, which are initially undefined. Initially some or all processes start execution. Any process can set a variable to a value, thereafter the value never changes, i.e. it is a single-assignment language. If a process tries to read a variable that is not set, execution is suspended until the value is set by some other process. This gives a dataflow mode of computation, where availability of data serves as the synchronisation mechanism.

If two processes try to write the same variable, an error will occur and the programmer

\(^1\)This is analogous to assembler usage of registers. A clever programmer can make programs run significantly faster by knowing how variables are used and allocating registers correspondingly, in practice this is usually not done because it is too costly in terms of programmer time, and because the work is often lost when a program happens to be altered.
will be notified. A process can not check whether a variable is set, it can only access its value. Hence correctness is independent of timing, and scheduling can be left to the system\(^2\).

### 5.3.1 Evaluation

Debugging such a system must be the dream of any bugcatcher; all behaviour depends on values of non-changing variables. A post-execution dump of variable values is all that is needed to re-execute any part of the computation, since values do not change and processes have to wait for them being set. Load-balancing and utilisation will pose some problems, any knowledge about preferable scheduling and process allocation can not be utilised. The good side of this is that it is in effect abstracted away; it will never need to be updated, and it will not cost any programmer time.

### 5.4 Bilingual Programming

We note this interesting observation by Carriero and Gelernter [CG90]; (their favourite pair is C and Linda):

> "Parallel programming requires the use of a computing language and a coordination language. Broadly speaking, the coordination language is the glue that allows us to build a unified program out of many separate activities, each specified using a computing language. A computing language (Fortran, C, Lisp, ...) allows us to compute values and manipulate local data objects; a coordination language must allow us to create simultaneous activities, and must allow those activities to communicate with each other."

Foster and Overbeek report experiences from this way of programming, where the sequential parts of a program is written in a traditional, sequential, "low-level"\(^3\) programming language [FT90]. The coordination is programmed in Strand.

The extensive use of a well known sequential language allows for efficient computation. All these sequential parts of code must be purely functional and without side-effects, i.e. they communicate with the rest of the world only through arguments and returned values. These returned values will be used for subsequent computation, as organised by the high-level control language. This further computation will not necessarily take place on the same processor, hence the value must be passed by value, not by

---

\(^2\)It seems to us that time-dependent non-determinism would be introduced if the processes could check if variables were set and modify their actions accordingly.

\(^3\)In these authors’ notation, C and Fortran are low-level languages, whereas Strand and possibly other newer languages are classified as high-level. The usage of the terms coordination versus computing language is also relevant.
The returned value depends solely on its arguments. Now a “high-level language” is used for writing the control structure of the program, which is easy because such languages have special mechanisms for controlling parallelism. User-defined data from the low-level language can be passed back up to the Strand layer and subsequently down to other low-level functions, but in the described implementation only simple data-types can be examined on the Strand level.

The strict discipline forced on the programmer through the functional style is seen to be an advantage in itself. Both the high-level and the low-level languages are used for what they do best, rendering efficient and clean code in both and “high-level control at low-level speed.” This programming style will typically lead to reusable libraries of sequential code, and clean reusable pieces of high-level code (or so the authors claim.) Since their high-level language performs automatic load-balancing, such programs will run unaltered on sequential and parallel computers.

Conservatives finding the dubious efficiency of previous “fancy languages” a major reason for distrust, may find solace in the fact that only a very small percentage of the computing time will take place in the coordination language. All inner loops are likely to be in the computing language. If a very small part of the code is coordinating, lousy efficiency in this part will not matter much.

The authors also point out that high-level concurrent languages are better suited for performance monitoring (than are “low-level” languages), since entry and return from sequential components offer natural events.

### 5.4.1 Evaluation

The fact that the sequential parts are purely sequential and can be debugged as such will enforce a multi-level debugging style, where sequential problems are removed first, these fit our definition of trivial debugging. Synchronisation and performance amelioration is left for high-level debugging⁵.

---

⁴References usually have no meaning in different address spaces. In case of a scheme with some kind of distributed shared memory, remote pointers or global object identifiers, the system could handle passing by reference, too.

⁵In this they agree with Garcia-Molina and Germano [GGK84, Section III a], see also Section 8.2.
Part II

The World at Present
This part of the thesis describes the state of the art of current multiprocessor monitoring and debugging systems.

Current monitoring and debugging systems depend a lot on the programming model, the programming language, and the rest of the programming environment. The bond between these factors and the debugging/monitoring system is often much more visible than the similarities between a set of debuggers, particularly when we consider research systems. Since each debugger system is custom designed for a specific programming model, language and environment, it is very difficult to construct a universally meaningful model with which we could compare all systems. We have therefore grouped systems with similar properties together and described each system in enough detail for the reader to see its applications, before evaluating each system.

The next chapter discusses some aspects relevant for monitoring and instrumentation in general, and is followed by a chapter describing real systems. Then follows a general discussion on debuggers for parallel system, and again a chapter with examples. The last chapter in this part mentions two other tools which need to be mentioned, but which do not fit in neither of the other two groups.
Chapter 6

Instrumentation of Multicomputer Systems

McDowell and Helmbold in their survey [MH89] summarise what researchers call monitoring and debugging:

"Monitoring is the process of gathering information about a program’s execution. Debugging [...] is “the process of locating, analyzing, and correcting suspected faults,” where a fault is defined to be an accidental condition that causes a program to fail to perform its required function. Since monitoring is often an effective procedure for locating incorrect behavior, it should be considered a debugging tool."

We agree, but note that debuggers for parallel systems are more tilted towards program understanding than “affirmative action”.

Burkhart and Millen [BM89] also see that the two processes may have much in common:

"Tools are needed that help the user understand the dynamic aspects of his program. From a programmers point of view there are two important motivations for a careful program-execution analysis: performance measurement and debugging."

Aral and Gertner [AG89] see problems with parallel programming, in effect coupling performance to correctness:

"Parallel programs [...] introduce a new class of bugs that are related to their dynamic properties. These certainly include time dependencies such as race conditions, deadlocks, starvation, and more subtle side-effects of synchronisation such as “fairness”. They also include errors which are not bugs per se but which contribute to poor performance. Such issues as
the poor scalability of the application program to many processors often violates the rational for having “gone parallel” in the first place.”

Some authors consider static analysis to be different from monitoring, but Gregoretti et al [GMZ86] use a fairly wide definition of monitoring:

“[Monitoring is] the process of extracting static and dynamic information concerning both a computational process development and its execution.”

Gentleman and Hoeksma [GH83] see reasons for adapting the terminology:

“The change of terminology [from debugging to “dynamic behaviour monitoring”] also helps in certain quarters where errors in programs are believed to be eliminated by proofs of correctness, and hence debuggers thought to be unnecessary - an argument which has no bearing on the need for tools to examine program internals while executing with actual data.”

Whilst eagerly avoiding a common phrase like “same thing, new funding”, we think this addresses an important underlying problem: debugging is often not seen as an area of serious research. This may explain the apparent lack of new developments and advanced commercial products in the field. Another reason for both of these phenomena may be that debugging in the general case is difficult to do!

Plattner and Nievergelt [PN81] point out that design, analysis and implementation of a program involve mental activities on two bases: program text alone or execution information. The theoretical focus is on the first one, but much work spent in the second. Conceptual foundations necessary to study process monitoring systematically have not yet been laid. Therefore use of monitors is an ad-hoc exercise if they are used at all. In their opinion:

“The most important aspect of a monitoring system is its monitoring language.”

Maples [Map85] writes about the MIDAS computer, which can operate as a parallel data-flow machine, and has the following to say about program understanding:

“Analyzing performance in a data flow environment is similar to monitoring fluid through a complex maze of pipes of different diameter. Although input fluid eventually comes out at the other end, its rate of flow through the system is determined by complex issues, particularly if valves in the structure are opening and closing asynchronously.”

Lazzerini and Prete [LP86] believe that sequential and distributed tools both have a role to play in the debugging of distributed system:
“The former ones include such facilities as the ability to examine and to modify variables, to set breakpoints and to invoke source level stepwise execution. The latter ones should provide monitoring and control of both processes and their communications.”

The point that the monitoring and debugging process goes through several phases is made by several authors. It would be nice and user-friendly, though, if tools could be combined to support a more than one phase.

On the ACM Workshop on Parallel and Distributed Debugging a panel set out too discover if parallel and distributed debugging were actually the same, or which was more difficult. A full agreement was apparently not reached. The discussion, according to a summary by Tom LeBlanc [LBPM88] seems to have been influenced by a lack of agreement on what a parallel computer is. Some of the participants seemed to classify only computers with shared memory as parallel; message-passing computers are then distributed computers with shorter wires. However, similarities and differences seem to exist. Debugging heterogeneous sets of processors is definitely difficult, and clearly distributed. The existence of inter-process communication is a factor present in both the parallel and the distributed case.

In this thesis, we are primarily interested in debugging and monitoring homogeneous message-passing MIMD computers. These typically have no shared memory, communicate exclusively through messages, have sets of heavy and light-weight processes on each node and exhibit significant non-determinism.

Beier and Bemmerl [BB88] have seen by experience that:

- Different tools must be properly integrated, i.e. use the same data and symbol conventions.
- All tools must support several abstraction levels.
- Tools have to be based on different instrumentation techniques.
- The tool environment must be independent of the node processor.
- The programming environment must be easily expandable.
- The tools must be portable.
- The system must have a friendly graphic/menu-based human interface.

### 6.1 What Parallel Programmers Need

Ahuja et al [ACG86] enumerate what programmers need:

- A machine-independent and (potentially) portable programming vehicle
• A programming tool that absolves them as fully as possible from dealing with spatial and temporal relationships among parallel processes
• A programming tool that allows tasks to be dynamically distributed at runtime
• A programming tool that can be implemented efficiently on existing hardware

Miller and Yang [MY87] and their system IPS (Section 7.3) emphasise two points: give the programmer a complete picture of a programs execution, and more than a tool that provides extensive lists of performance metrics; users need tools that will direct them to the location of performance problems. Data should be provided both from intra- and interprocess level, programmers should be provided with maximum information about their programs. There must be a logical and intuitive organisation to the data, and its presentation. The programmers should be provided with answers, not numbers. It should be possible to detect a situation where cost is dispersed among several procedures, and across process and machine boundaries, and also to determine the effect of contention for resources.

Moser concludes [Mos90] that analysis of parallel and distributed systems is too complex to be trusted, empirical experiments and measurements are necessary.

Bemmerl leaves no doubt [BLT90]:

"The behaviour of multiprocessors with several concurrent processes is only understandable by collecting a great amount of runtime information."

In [Bem90], he underlines that adequate tools for all phases of the "software life cycle" must be at hand. He states that the first phases or this cycle are fairly similar for sequential and parallel systems, and that an effort to improve the field of parallel programming should be directed towards the later parts, e.g. validation and analysis.

"Apart from these monitoring and analysis tools, the programmer of a parallel computer needs in particular support for the mapping of the logical program structure onto the physical parallel architecture."

6.2 What is Monitoring?

On general multi-user systems, we find a very important difference between program profiling (as with prof) and machine monitoring (as with perfmon).

A profiler typically generates a post-mortem summary of program execution. Profiling is requested when a program is compiled and linked, including calls to profiling code. This code records information at run-time, which can be summarised to make information more accessible. Typically, profilers inform about the number of invocations of functions, sometimes also about basic blocks. Advanced profilers include information
6.2. WHAT IS MONITORING?

about where a function was called from, and what other functions it called. Sometimes profilers also give information about the time used in different calls. Program profiling is focused on what one program, or a part of a program, does. Profiling is abstracted from the state of the computer as a whole, e.g. the program profile should not depend on the state of the computer.

Machine monitoring gives the user a “holistic” insight into the state of the computer as a whole, which depends on what each program does, and on their cumulative effects, such as system buffer consumption, internal queuing, virtual memory and disk usage. When needing system statistics from a single program (such as total run time), users commonly run on computers known to have nothing else to do.

With single-user systems, such as personal computers, this distinction is less relevant, and with single-process systems totally irrelevant. As we have tried to argue earlier, we believe that a very important and interesting class of the “real” multi-computer work being done is the one where a single job to be done is so large that the user needs a parallel computer to get her results in reasonable time. Many of the currently available multi-processor systems are space-shared, letting several users have some nodes of the parallel computer at a time, but not time-shared. Then the distinction between program monitoring and system monitoring again becomes somewhat blurred.

We have given examples of a brief selection of programs of this kind in Appendix A. The selection reflects the authors’ personal preferences; many systems exist without us using or crediting them, they may even be better than these. Life is to short to try them all!

Invasiveness is also called the probe effect [TCO91], and is the term used to denote the phenomenon that modifying a system for monitoring or debugging alters the execution. Invasiveness can have at least two effects. Reduction of system performance is common, and need not alter the path of execution. From a debugging point of view, slight reductions in performance are common for various reasons, and do not really matter. Invasion may also result in a distortion of the ratio of the time needed for different tasks. In systems with non-determinism, any invasion into a race\textsuperscript{2} may result in dramatically altered execution paths\textsuperscript{3}.

Partial solutions are readily at hand: hardware monitors allow us uninvasive monitoring [BLT90, TFC90, Pla84]. Unfortunately they also tend to be expensive to construct, and very hardware dependent. Frequently polling interrupt-driven software monitoring systems can provide reasonably exact information, but may reduce the throughput of the system significantly. Beier and Bemmel [BB88] claim that nonintrusion is more

\textsuperscript{1}Which is one of the reasons that test programs are often carried out at night, and that computer scientists may be observed at work at odd hours.

\textsuperscript{2}Races are situations where two participants try to access a shared resource simultaneously, and their order makes a difference.

\textsuperscript{3}Imagine that we live on a node with two neighbours. We will choose the node talking to us first as partner in a major effort involving two nodes and never talk to the other again. Both these nodes send us a message, at exactly the same time. At some granularity, one message will be received before the other. Based on a minute initial difference we see a major variance in the result. The system is unstable.
important for performance monitoring than for debugging. Therefore they use a software monitor for debugging and a hardware monitor for performance monitoring. Gentleman and Hoeksma [GH83] find hardware assistance of “high-level debugging” useful, since it provides access to information not otherwise obtainable, and allows execution to proceed at full speed at debug-time, thus not disturbing timing-dependent phenomena. They suggest that it can be implemented by translating a high-level criteria specification to a low-level equivalent.

Although one tries to reduce the effects of it, software monitoring will always affect execution. The choices with software monitoring is to make this invasion as slight as possible, or to live with it. If nondeterminism is removed from the system, invasion will not really matter, as it can not change the course of the program. Very few real parallel systems are nondeterministic, and HC16 certainly is not. The PIE system [LSV+89] deals with invasiveness by time stamp modification. The time stamp of an event is adjusted according to the events occurring before it, and time consumed doing monitoring work. This can not be guaranteed to work.

6.3 What to Monitor?

Within the general framework of the monitoring system it should be possible to monitor any resource of the computer. The choice of what to observe seems somewhat on the side of this thesis, let us just remark that any information collectable by the operating system could be shown to the user. Ideally, the parameters displayed should enable the user or programmer to understand the execution, and if necessary make meaningful modifications to her setup and program.

For example, the top program by William LeFebvre [LeF] under UNIX displays general information about the system: the last process id assigned, load average for the last one, five and fifteen minutes, current time, total number of processes, number of processes in each the state (sleeping, abandoned, running, starting, zombie and stopped), and the percentages of time spent in each processor state (user, nice, system and idle), memory usage (real, virtual and free). For each process among the top consumers, it displays identity information, priority, size (of text, data, stack and resident memory), state, time used (system and user CPU seconds) and some weighted CPU usage information. This is a typical machine monitoring program, and enables the system manager and enlightened users too see usage level and potential bottlenecks on a per-process level. The reason for the name of this program (top) is that the processes are sorted, biggest consumers first, giving a very clear hint about which process to do something about if something must be done.

Another machine monitor, xperform in addition to some of the metrics above also gives information about disk transfers, interrupts, input packets (over the network), output packets and collision packets. This tool only presents summarised information, and thus leaves more work to someone wanting to correct a problem.

Tsai et al [TFC90] have found the following events interesting on the process level:
creating and terminating processes, process synchronisation, interprocess communication and external signals.

Bemmerl et al [BLT90] generalise into three classes of events:

**Execution events** including execution of a particular statement (first or n-th time), entering leaving procedures, OS entering scheduler, page-fault handler, or communication procedure,

**Data event** including setting or accessing variables, and

**Parallelity events** including communication, queuing operations, semaphore usage, etc.

For monitoring, the parallelity events are the most interesting, execution events are typically used for debugging, see also Section 9.3.

### 6.4 How to Collect Information

There are three fundamental ways of collecting information about execution, hardware assisted monitoring, software monitoring and hybrid solutions. According to Gregoretti et al, the sensors perform three functions [GMZ86]:

**Detection** i.e. discovering that an event has occurred. Detection may be active and lead to some action, or passive.

**Isolation and filtering** i.e. determining what to do when an event is discovered.

**Notification** i.e. reporting to the external environment that some event has occurred. Storing the information is one variant of this.

These functions are closely connected. Only when an event is detected will there be a need for isolation or filtering considerations, then notification may or may not be invoked. The choice of sensor mechanism is basically a compromise between non intrusiveness and observability\(^4\)

Information collection can be based on four principles [BHL90]:

- **traces** store information about time and type of event,
- **counters** send numeric information when an event occurs at regular intervals, time is implicit by reception,
- **timers** measure time between events, send information,

\(^4\)Observability [GMZ86]: “The capability of the sensor mechanism to take advantage of the knowledge it has about the object or the event under observation.”
integrating counters use pair of registers where one stores instantaneous values of a parameter, and the other accumulates, e.g. keep the length of a queue in register one and add this to register two at fixed intervals. Polling register two will then give information about the state of a resource.

6.4.1 Hardware Collection

Bemmerl et al [BLT90] share an experience from the TOPSYS project, which uses several kinds of monitoring techniques:

"The only but great advantage of the HW-monitor is the nonintrusiveness."

This is very important, because as Plattner and Nievergelt [PN81] clarify:

"In real-time monitoring, either the target process is not delayed at all, or else the cumulative delays are unbounded."

Letting a processor handle monitoring will imply some delay, unbounded cumulative delays are of course not acceptable in real-time processing, therefore, real-time monitoring implies a separate monitor processor! In our view, approximate data can be useful too, especially if the choice is to have approximations or no information at all.

According to Gregoretti et al [GMZ86], information can be collected at three different points in the machine:

Inside CPU:
This is a very expensive and complicated solution, but the only one which will render all information about the program available, since registers, internal caches, and prefetchers are only available here, and becoming ever more important. External information blocked by surrounding logic is not registered.

Outside CPU:
Collection can be done on the CPU output pins, this is technically less demanding since it does not require debug versions of the processor, but still processor specific.

On communication channels:
When collecting information on the buses of the system, all internal information is lost, but external information is accessible.

Tsai et al [TFC90] tell us that monitoring is more complex on real-time, distributed computing systems than on centralised, sequential systems. Since both the real-time aspect and distribution are generally accepted as complicating factors, it is hard not to agree. The reasons, we are told [Ibidem], are the presence of multiple, asynchronous
processes, critical timing constraints, and significant communication delays. Since
timing in real-time systems is critical, monitoring methods which introduce delays
are not acceptable. All software monitoring methods will introduce delays, this leaves
hardware monitoring as the only solution. The authors of this article suggest moni-
toring the communication channels.

6.4.2 Software Instrumentation

Software instrumentation can be implemented in several ways. One way is to augment
the run-time libraries so that they collect information when they are invoked. This
is very accurate, the timing and numbers will be exact. It is however limited to the
system calls. The kernel can also be augmented so that information is collected, one
example of this is when the scheduler saves execution data.

Another way is to modify the users code. This can be done by manipulating the source
code, the object code, the linked code, and typically takes the form of putting in extra
statements. User code manipulation has the advantage that it can be restricted to
specified parts of the code.

Another common technique is to poll the program counter at fixed intervals to de-
termine where execution takes place. This is cheap as long as the polling is not to
frequent, but then also not very exact.

6.4.3 Hybrid Solutions

Software and hardware instrumentation can be combined, giving hybrid solutions.

Gregoretti et al [GMZ86] mention one way where the detection is executed by software,
and all other functions are done by hardware. With extra hardware this will improve
performance and reduce invasion, compared to software instrumentation.

The other obvious solution is to reverse the roles: let hardware to the detection and
invoke software functions. This is for instance what happens in the Intel 80386 with
the use of debug registers: the programmer has three special registers available, when
the addresses specified in these registers are used or modified, an interrupt is raised.
This interrupts can be handled by a specified function.

6.5 How to Display Information

Alva L. Couch [Cou88] writes about graphical presentation of program performance.
He splits display types into two groups, dynamic and static ones. The significance of
the grouping is not quite clear to us, but it seems that static displays are those not
being updated, whereas dynamic displays “live”, at least this usage of these terms
seems reasonable.
His dynamic displays show instantaneous information. They are essentially abstracted pictures of some aspect of a parallel computer, where colouring and sometimes annotation of symbols provide information. A typical example of this would be a graycode circle like the one on the front of this document, with the colour of each node showing CPU activity (more active could be shown as darker, or more red), and the colour of each connection similarly indicating the amount of message traffic. Any graphical representation of a computer could be used for this. Information can also be presented by one or more bars per node, or indeed in any other chart type which the graphics interface can display.

One display which seems particularly interesting for communication is the point-to-point matrix, where each processor is represented by a line in both the vertical and the horizontal direction of a matrix. The colouring of the intersection of the lines will indicate the amount of traffic from one to the other. Note that each pair of nodes will have two intersections, one for each direction of communication. This display may be "dumped" for compactness, i.e. nodes $a$ to $b$ can be represented as a group, instead of as individual nodes.

The static displays show information over time, or as summaries. Examples of this can be the amount of communication per time as a strip chart with the axes giving time versus amount, and the accumulated number of for instance messages of varying length with the axes indicating message length and accumulated count.
Chapter 7

Current Parallel Monitoring Systems

7.1 Observer

Roisin and Santana [RS91] discuss what they call observation tools, which are not traditional monitors, nor do they seem to be debuggers:

"The goal of such an interface is to demonstrate and to explain the inner working of an application, i.e. to show how the different entities which compose an application evolve and interact during the execution.

Their Observer is a tool for observing object-oriented distributed applications in Guide. It has three purposes: as a teaching tool to give the user understanding of algorithms, as a debugging tool to allow graphical debugging (see invocations and stacks), and as a demonstration tool. The latter motivation is the main one, because:

"Demonstrations have become an important matter in the software application area, in order to convince the clients that an application matches its specifications or that the ideas and concepts it shows are worthwhile."

The observer provides information on a high abstraction level, with icons for activities, objects, and object classes summarising the state of execution. Typically the user will have windows for each class, showing some or all of the objects in them. She can also look at the “activity stack”, inspecting chains of invocations.

7.1.1 Evaluation

This system is very application oriented, and uses the abstractions from the underlying, object-oriented system. This function is very different from what we call
monitoring, defined and discussed in Section 6.2, but these tools may in some cases give the programmer a similar kind of insight into the program behaviour.

7.2 MAD

Rubin et al [RRZ89] talk about their system for Monitoring, Animating and Debugging, or MAD. The main aims of such systems are non-intrusiveness, interaction, and user-friendliness. The authors consider non-intrusiveness crucial, and claim that it can only be achieved by augmenting hardware. MAD focuses on Interesting Events, which are changes to the program or system state. Some of these are system generated and some are inserted in the code or specified at run time. Compound events combine other events, with each event there is a count and an active flag.

Events can be connected to views, which display information visually, this has proved to be very interesting. It seems that MAD is primarily a program understanding tool.

7.2.1 Evaluation

Since the authors augment hardware to avoid intrusion, we are a bit surprised that they in other cases allow code modification. Run-time specifications may be non-intrusive, but only if they are executed on auxiliary hardware.

7.3 IPS

IPS\(^1\) [MY87] and IPS-2 [YM89, MCH+90] are performance monitoring tools for processes communicating with messages (IPS-1), later expanded to multiprocessors (IPS-2). Their aim is to guide the programmer to the location of a performance problem, and describe the problem in terms of the source program. The tool should provide answers, not just numbers. These tools work in two phases, data collection and data analysis.

IPS-1 consists of three major parts, which are all distributed among the individual nodes in the system.

**Agent:**

The agent is a set of data collection primitives, in the kernel and run-time routines. All data are always collected, the user has no possibility or need for affecting this. Due to this strategy there is no need for running a program again to get information not collected in a previous run, the drawback is increased storage demand.

\(^1\)The name IPS looks like an acronym, but it is not clear what for. My guess would be something like Interactive Performance monitoring System.
7.3. IPS

Data pool:
Information from the agent is deposited in this pool. It is stored as raw data, and intermediate results for the process and procedure levels described below. Since this pool is distributed over all nodes, no network bandwidth is wasted on writing to it.

Analyst:
At least one master analyst, providing an interface to the user. The master coordinates information from the slave analysts on the individual nodes, the slaves preprocess data for presentation, thus parallelising this task to increase performance. The master analyst keeps information cached for the machine and program levels.

IPS-2 consists of four major parts, the agent is replaced by or renamed to instrumentation probes, and the functionality of the analyst is split, so that the new analyst just analyses, and the user interface is a separate part. The data collection in IPS-2 comes from two sources, the run-time routines and modified call hooks generated for gprof.

The system keeps intermediate results tables for metrics at the process and procedure level in each slave analyst, and passes them on when requested. Result tables for the machine and program levels are always built in the master analyst.

IPS presents information on different levels:

Program:
On this level, the parallel computer is seen as one integrated resource. Information is synthesised from information on the individual nodes, and is cached in the master analyst.

Machine:
Here, machine means node, and summary information is displayed on a per-node level. The node may not be a logical layer between the program and the processes, but it is still included because systems can specify allocation, and allocation is often one of the major performance considerations. Information is presented without no structure of the activities on the node.

Process:
A distributed program is seen as a collection of cooperating processes. Information can be shown for processes on each node, or over the nodes of the multicomputer.

Procedure:
This gives detailed information of the behaviour of each process, necessarily local to a node since processes are.

Primitive activity:
This level gives information about the primitive events of the system. Events
on this level can be mapped to information on the higher levels, it is not clear to me how. The primitive events are stored with local time, and are: process start, process end, process blocking, process un-blocking, procedure enter, procedure exit, message send, message arrival, message receive, and attempted receive.

In IPS-1, events on the procedure level were collected by sampling. In the second version, even these were traced by generating events as they are performed, thus greatly increasing precision.

Yang and Miller [YM89] observe two gaps in the current state of monitors:

**semantic gap** between the semantics of the structures of programs and the semantics of viewing systems: people use complex and well-defined methods for creating applications and ad-hoc performance evaluation systems.

**functional gap** between what the users get, usually low-level information, and what they need: guidance as to how to improve performance.

From version one to version two of IPS, the performance was improved significantly through reducing the size of traces and memory-mapping requests for time. An added graphic user interface also improved usability significantly.

A special feature is the **critical path analysis**. The system identifies the path through the execution which consumes most time, if the speed of this path is improved it will improve the speed of the program as a whole. This information is based on a program activity graph constructed from the traces.

Later versions are planned to include second and following most critical paths. The system identifies the individual parts that occur most frequently in the critical path, and also common sequences of events.

Another novel feature is the **phase behaviour analysis**, where the system tries to identify the phases of programs. These phases are typically to some extent independent, and can be ameliorated individually. When phases are identified, other information can be restricted to one phase.

### 7.3.1 Evaluation

That the user has a choice of abstraction levels is a nice feature of this system. This tool seems not to address the intrusion issue, i.e. to which extent the presence of the monitoring system affects the execution.

Replay is not necessary under IPS, since all information is stored in each run. This is a strong benefit, since it allows the programmer to look at different aspects as her

---

2The authors use the term _performance measurement tools_.

3Instead of using an expensive system call to find the local time, the system maps the clocks into the users address space. This is a memory location and can be accessed very quickly.
understanding evolves, without introducing distraction and confusion through a new execution. It also saves the re-execution time.

7.4 PIE

The Programming and Instrumentation Environment (PIE) [DHE89, LSV+89] focuses in on target performance levels, which is considered absolutely essential. It is

"a theoretical framework for developing techniques to predict, detect and avoid performance degradation."

The effort goes in two directions, firstly avoiding performance bottlenecks through combining a coding methodology and performance prediction models early, secondly, performance debugging. The environment consists of a multiprocessor version of C, an augmented editor for automatically evaluating and modifying programs, a monitor for collection and storage of event data⁴, an information manager combining development-time and run-time information, and a graphical user interface for viewing the information. The last part is necessary, staring at numbers is a very inefficient way of approaching problems in this field. As noted earlier (Section 6.2), we must not confuse profiling data and performance data from the computer. The PIE people agree in this:

"If programmers are to know what computations do at particular times, they need to be able to visualize executions on the level they program in."

PIE provides many fascinating kinds of graphics, showing for instance state of execution over time and parallelism achieved over time, and it shows explicitly what blocked processes are waiting for. All displays are augmented by textual information.

7.4.1 Evaluation

We are convinced that this approach of avoiding bottlenecks instead of removing them when observed, is a very sensible way to go, since all "Common Wisdom" in systems development points out that detecting algorithmic flaws during system testing is very expensive.

It is also an important point that the programmer needs to get the run-time information related to the source code in some way, that makes it a lot easier to understand why some phenomenon occurs.

The choice of sensors removes the intrusion problem.

⁴This data is supposed to be collected by software, hardware and “hybrid” sensors, in 1989 only software sensors were in use.
7.5 TOPSYS

Bemmerl [Bem88] introduces the Multiprocessor Monitoring System (MMS) and claims that (up to 1988):

“No adequate tools are available for debugging, performance measurement and visualization of program execution.”

The problems with current systems vary. Some tools operate outside the programming environment, i.e. they operate on too low a level and do not provide the programmer with information on the same level as her programs. Others are more batch oriented, forcing the user to modify source code, or they are architecture-specific. These are clearly too complicated to use.

“MMS is usable for program development, program optimisation, for studying of parallel computers and concurrent programs.”

Bemmerl [Ibidem] tells us that MMS offers the following:

- A window-based concurrent debugger, with features for displaying and modifying the states of programs running on the multiprocessor, including breakpoints and tracing.
- A performance analyser to localise bottle-necks.
- A visualisation tool gives graphical representation of the multiprocessors state space.
- Expandability with batch-oriented tools
- User friendliness, graphic support
- Different abstraction levels
- Portability, integration, expandability, adaptability

Beier and Bemmerl [BB88] describe SWM, which is a software monitor, and compares this technique to hardware and hybrid monitoring. This project seems to be a predecessor or a part of MMS and TOPSYS (to be described).

The different tools of SWM offer the programmer the possibility of specifying predicates and conditions at several abstraction levels. On successful evaluation, the tools can initiate various actions. Predicates can be specified over the control flow, data flow, and concurrent objects. The programmer can also manually execute actions to inspect or modify the state of the program. The predicates are evaluated by automatically inserting statements into the user program.
Bemmerl later [Bem90] gives an overview of the TOols for Parallel SYStems (TOPSYS) project. This project group searches for a methodology for gaining productivity in parallel software development. The hierarchy of the tool system is well defined, and a user can choose to look at any level she wants. In addition to PATOP and MMK described below, TOPSYS includes a specification and mapping tool called SAMTOP, a debugger called DETOP, and a graphical animation tool called VISTOP.

The TOPSYS project places great emphasis on portability and flexibility with respect to the processor, topology, programming language, process model, compiler, operating system, tools, monitoring techniques, tools and abstraction levels to be supported. The designers realise that meeting all of this will be too tall an order. Generality is achieved by implementing all tools on top of a common distributed monitoring system.

The system uses four types of monitors: simulation, hardware monitoring, software monitoring, and hybrid monitoring. These offer the same functionality, and are thus exchangeable.

The Multiprocessor Multitasking Kernel (MMK) [BL90] has integrated dynamic load balancing. This system concentrates on problems arising from the distribution of resources, especially memory, and tries to investigate virtual global memory on a multi-computer. The article states some measurement data, concludes that

"MMK is an adequate compromise between the hardware oriented operating systems and programming concepts with high abstraction levels like Linda which often do not provide efficiency for numerical applications."

The system integrates global object space and dynamic load balancing invisible to the programmer, simplifying the use of multiprocessor supercomputers.

Their hardware monitor system [BLT90] has a separate monitor board connected to (each?) node with a cable. This system's hardware monitor is inserted between the CPU and the board, i.e. using Outside CPU monitoring. Since the system needs access to the program counter, which is not available outside the CPU, this counter is emulated in the monitor. This project uses a "bordout" version of the processor, to get access to other information not usually available outside the CPU.

The motivation of the Performance Analysis Tool for Parallel systems (PATOP) [BHL90] is to discover performance bottlenecks which can be caused by overloaded communication links and inappropriate workload distribution. The main usage of PATOP is to optimize parallel programs and their mapping to multiprocessor structures. The authors state that:

"There are three fundamental circumstances that lead to performance bottlenecks:

- Permanent overload of system resources due to a concentration of requests to a single system resource
- Permanent overload of system resources due to inefficient programming of sequential program parts"
• Short term overload of system resources due to synchronised requests by parallel program parts."

Short term overloads are characterised by possibly long delays but low average usage of the resource. Permanently overloaded units also have high average usage. The system needs the following properties:

1. Describe delays at system resources
2. Describe average usage of system resources
3. Enable the user to view the program at different levels of abstraction and granularity

Evaluation of events can be done immediately or later.

Presentation of monitor information must be available on several levels. PATOP supports the following:

System:
Shows information summed over all participating nodes. The granularity can be low, as no relationship between the individual components is seen.

Nodes:
Here the performance of the individual processors and communication links are shown, including information like idle time, time waiting in ready queue, traffic rates and delays over communication links, and delays incurred at send and receive calls.

Objects:
This level includes the statistics mentioned under nodes, specified per task.

Program parts:
This is a finer still detailing of resource consumption, not yet implemented.

7.5.1 Evaluation

This is a very complete system, including all tools from monitoring and debugging, to visualisation and dynamic mapping.

The auxiliary hardware for monitoring should remove the intrusion problem, and the choice of abstraction levels gives the programmer the possibility of focusing in on relevant parts. It is however not quite clear that the programmer would need or want such detailed control, given the dynamic loadbalancing?
7.6 Faust

Guarna et al [JGJM89] present Faust, the name was “chosen with no underlying acronym or rationale”. Their goal is:

“To design and implement a set of new tools specifically designed to help develop parallel programs. [...] To integrate these new tools with existing tools [...] and to ensure portability.”

The concepts of project databases and automated management seem central to this group. Programs can be statically analysed and the information seen in the context of the program through advanced tools. Traces of executions also go into the database, and can be viewed on different levels with options of filtering out uninteresting events.

7.6.1 Evaluation

This system addresses the question of what to do with performance data once they are available. The complete integration of tools seems to be more distinguishing than their actual tools up to this time.

7.7 SIMPLE

Mohr [Moh90] describes the system SIMPLE (Source related and Integrated Multiprocessor and -computer Performance evaluation, modeLing and visualisation Enviroment). This event-driven system has separated the processes of collecting information and processing it. Collection is done by the Zählimonitor 4 (ZM4), which collects local data and through the use of a data channel and a clock tick channel collects and synchronises them. Due to the resolution of the clock tick channel, the system can monitor systems up to one kilometer apart.

These traces are stored in a format which can be generated also by other tools, and can be filtered and clustered. The subsequent analysis is able to handle missing local trace segments, and can generate interactive and static graphics, and even animation.

7.7.1 Evaluation

The separation of collection and processing of data makes the intrusion due to monitoring smaller, that the monitoring agents are auxiliary processors removes it entirely. The integration of the traces is a very interesting task.
7.8 PEM

Burkhart et al [BEK+88, BM89] describe their Program Execution Monitor (PEM). This is part of the project $M^3$, see also Section 4.13.2. PEM consists of a monitor base and a family of tools which can be used isolated or in combination. The monitor base has four interfaces: high level language symbol tables, operating system tables, user interface and agent interface. The separate tools consist of agents residing between the monitor base and the processors, using the monitor base for communication with the user. Some examples are given of such tools:

- The *trap monitor* allows the user to define breakpoints at the programming language level, and to associate programs with these traps. Prior to running the requests are translated and installed. When a trap is encountered at run time control is either given back to the executing process, or the process is halted, awaiting interactive manipulation. This involves only the agent installed on the node.

- The *mailbox monitor* shows pending sends/receives on a communication channel, recording this number over time$^5$.

- The *bus count monitor* records information about conflicts in the usage of a shared bus. This is based on a hardware monitor of the bus, which calls a piece of software to handle its findings.

- The *idle-, access-, and conflict-monitors* support performance analysis.

- Finally, for conventional program analysis the system offers pc-histograms, system events and a timer.

The authors distinguish between different sources of loss (see Section 2.2), and PEM plots out a graph where the individual sources of performance loss are shown.

7.8.1 Evaluation

The pre-installation of traps reduces the intrusion problem. The choice of presentations of data makes them accessible to the programmer.

7.9 ELAN

The system for *Efficiency Loss ANalysis* (ELAN) described by Moser [Mos90] is a successor and extension of PEM and IMPACT. This system not only shows efficiency

---

$^5$Receives can be seen as negative sends, or vice versa, thus only one number is needed for communication channel status.
loss, but also distinguishes between different kinds of loss. Overhead is calculated by comparing traces from different configurations.

The fascinating part is that performance results are automatically entered into a spreadsheet which is an integrated part of the environment. These values can subsequently be presented to the programmer or even used by configuration tools to set up for better performance in later executions!

Moser concludes that:

"... a lot of effects can *not* be predicted or calculated. Parallel systems and programs are simply too complex and often depend on too many parameters. We have to do empirical experiments and measurements."

### 7.9.1 Evaluation

The use of a spreadsheet is an elegant idea. No building of custom tools required, the spreadsheet makes available a wider range of functions than one would care to implement for oneself.

### 7.10 Express

ParaSoft has a family of supporting tools under Express (Section 4.5.5):

**aspar**: an automatic paralleliser for C and Fortran, generating code for shared or distributed memory parallel computers,

**vtool**: a tool for visualising the memory access patterns of sequential programs, allowing stepping forward and backward through source code, useful for building up knowledge about an application,

**etool**: an event driven profiler, showing OS, communication and user-defined events over time,

**ctool**: communication analysis, showing the accumulated cost of various activities over the participating nodes,

**xtool**: source code profiler, giving time spent in routines and number of times statements are executed, providing information on the subroutine, source line and machine instruction level, and

**F90**: parallel Fortran.
7.10.1 Evaluation

Express is a commercial system, and seems to offer the programmer a wide variety of analysis tools. We do not know much else about the system.

7.11 Others

Abstreiter [Abs90] splits between real-time and post-mortem analysis, and chooses the post-mortem version for his system. Programming in this system is based on ParMod, an extension of Pascal, similar additions are also available in C and Fortran. The system lets the programmer detect programming errors and bottlenecks, analyse deadlock and starvation situations and tune a program with the aim of optimisation.

The system has two levels of abstraction: program observation and machine observation, i.e. what we call profiling and machine monitoring. All information relevant to parallelism is stored at run-time, interpreted afterwards. Program observation split into six classes:

- Program communication (time of communication),
- Program commEx (summary of communication and execution)
- Program commAmount (which tasks communicate, how often, how much)
- Program execution (number of executing tasks),
- Program load (how long objects have in in states (e.g. runnable))
- Program synchronization (duration of synchronization states)

Hardware observation provides communication, commAmount, execution and load.

The BBN offers the tool Gist [Te91]:

“to give the user a picture, in real-time, of the state of all processes running in parallel. It is used to discover bottlenecks in the code, whether these arise because of synchronisation problems due to processes awaiting data, or are caused by sections of the program which are especially CPU intensive.”

Joyce et al [JLSU87] observe that users nearly always need a variety of tools during the development of an application system, ranging from detailed, low-level tools to highly abstract application-specific tools. To accommodate this, their Jade separates detection and information collection from the task of analyzing and displaying information. Jade (or really Jipc) processes can (but need not) be loaded with monitorable libraries. Monitorable events are operations that may have an effect outside
the process where they take place, these are merged into an event stream, by being sent synchronously to a controller. This controller must reply before the process may continue, which is similar to the use of timesteps in PanParallacea.

The system can generate textual and graphical images of event streams. The article concludes that: collecting information with the same mechanisms as application programs simplifies life, collection of information should be separate from analysis, a variety of tools is needed, textual and graphical displays offer roughly orthogonal views into a system's execution, the ability to interactively control nondeterminism and reproduce specific computation paths is crucial, animation is effective, a monitoring system should exploit semantic information about an application system.

Burdorf et al [BFMP89] have observed that traditional profilers are insufficient for parallel systems. They have developed a tool that collects the following information: where CPU time is expended, quality and quantity data passed between functions, how much global data is referenced and modified, and how these characteristics differ among processors on the network. The system is used to evaluate LISP programs, run-time and post-mortem. Data collection can be interrupt-driven, i.e. at interrupts sample the program counter and identify the active function, or invasive, i.e. wrap code around functions to store information. Invasive collection affects the execution severely, and should only be used on the most critical functions. Interrupt-driven collection involves periodically invoking a timed interrupt which inspects the program counter, apart from finely tuned races, the interrupt handling should take the same amount of time on all nodes, and not distort execution. The authors claim that the wrapping method is unsuited for statically typed languages like C and Pascal, due to the lack of dynamic type information at run time. This restriction seems to be due to their fairly detailed definition of data collection, consisting of data types of messages.
Chapter 8

Parallel Debugging

Gentleman and Hoeksma [GH83] distinguish between debuggers and debugging tools, in their view a debugger is one of many debugging tools, and one that can:

".. dynamically analyze a program in a controlled environment where “breakpoints” can be set."

Bates and Wileden [BW83] use a rather different notation, to them a debugger is:

".. usually a person directing the error search."

Gray [Gra85] calls this person a bugcatcher. This seems appropriate, it has some of the derring-do feel to it that is characteristic for the people fearlessly wrestling with the finer details of split-second timing and very expensive machinery.

Garcia-Molina and Germano, Jr. [GGK84] tell us that for distributed systems:

"The correction of the software (after the bug has been identified) [is] relatively straightforward. The problem is identifying it."

This also seems to be true for parallel systems, and often even for sequential ones!

Araki et al [AFC91] see the first problem of the bugcatcher to be what information to provide. After all, she herself must set up the debugging system to describe useful aspects of the computation, and at the start of the debugging process the feeling of not knowing what you look for is common. When presented, this information must also be comprehended. This is often unnecessary complicated because the abstractions of the programming system are not supported. In their view [Ibidem], debugging is a process consisting of localisation and correction. In the localisation phase, the bugcatcher formulates a hypothesis about the bug, which she then proceeds to try to verify. If the hypothesis is shown to be correct, further analysis and correction can take place. Otherwise the knowledge gained from the refuted hypothesis contributes

107
to the programmer's understanding of the system. The authors try to convince the reader to go through a process of simplification, narrowing and modification of the initial hypotheses.

Gould [Gou75] analysed the debugging process, and found that most of the participants in their study seldom used interactive debugging tools. This may still be so in the 1990's, and for several reasons: debugging tools are not always easy to use, and even if they are well made the programmer has to get to grips with yet another interface and model. Also, the programmers may doubt the necessity of debuggers.

Thomas G. Moher [Moh88] agrees that debugger usage is too low:

"Specialised debugging tools are widely available but infrequently used. The ironic challenge for debugging researchers is to invent tools which will bring programmers back to the computer."

Peter Bates defines [Bat88]:

"Software debugging is a process in which debugging tool users synthesize selected artifacts of program execution into models that reflect the actual behavior of a system under investigation. [...] Debugging tools facilitate modelling by providing support for monitoring a system under study and experimenting with its future behaviour."

Moher [Moh88] tells us that debuggers have typically offered services in three classes: process visualization, state selection and process modification. The two first help identifying problems, the latter improves understanding of its consequences. The main point of most debuggers for parallel systems is to help isolate trouble-spots. The options of setting values and altering the path(s) of execution often seen in sequential debuggers is not a central issue in parallel debuggers.

McDowell and Helmbold [MH89] argue that parallel debugging should be more than a set of traditional debuggers connected to a set of single node programs, however that is exactly how most of the current so-called parallel debuggers work.

Aral and Gertner [AG89] also give harsh judgement:

"Although some advanced debuggers provide a programmer with the ability to define complex logical assertions, they are inadequate for debugging parallel programs due to the high overhead of monitoring these assertions."

We have to agree. It seems typical and reasonable, though, that the development of debuggers is delayed relatively to the systems to be debugged.

Below we will look into some of the complicating issues of debugging parallel computers and then some of the systems available, but first we will discuss how to specify criteria in a debugger.
8.1 Debug Predicates

Miller and Choi [MC88] define different types of predicates for specifying breakpoints in a debugger. Their classes are: simple, disjunctive, conjunctive, linked and distributed predicates. Their simple predicates are what we know from sequential programming. The others are discussed below, in Section 8.3.

Bemmerl [Bem86] generalises about predicates and actions. Actions are classified as conditional or unconditional. Predicates are split into five classes:

Execution predicates:
Predicates about execution of code, like invocation of or return from functions, count of number of times a statement is executed, etc.

Data predicates:
Predicates over the values of data.

Exchange predicates:
Predicates concerned with kernel operations, e.g. creation of processes, manipulation of semaphores, message sending or reception.

Assertian predicates:
"General program flow conditions", these typically include execution, data and exchange predicates, and may involve evaluating priority of tasks, value of semaphores etc.

Combination of predicates:
Any logical combination of the above-mentioned predicates, in addition to the relations “before” and “directly before”.

Since we will go on to integrate “common” and performance debugging, we need to add one item to the list:

Instrumentation Predicates:
These are predicates concerning performance and resource consumption.

8.2 Parallel Debugging Problems

Joyce et al [JLSU87] enumerate problems with parallel debugging, testing and evaluation:

1. Many foci of control complicate the use of program counter and process state.
2. Communication delays make it difficult to determine state at any one time.
3. Inherent non-determinism.

5. Interactions between system and developer may be difficult.

Fowler et al [FLM89] see three issues complicating parallel program analysis: non-repeatable behaviour, interactive debugging distorting execution and the enormous amount of data involved. We still focus in the internal state if the individual processes. Since that is not where the problem lies, we are told that this focus must be changed to the interactions between processes:

"Tools for analysing single processes, such as symbolic debuggers, are insufficient and must be augmented with facilities that enable analysis of interprocess relationships, including communication, deadlock and resource contention."

Garcia-Molina and Germano [GGK84] see four reasons for distributed debugging being more difficult than sequential debugging:

1. multiple asynchronous processes: The systems have various loci of control, making them harder to program and understand. Bugs are often sporadic and difficult to reproduce.

2. multiple processors: a problem which is easy to spot on a single processor may be more hidden in a multi-processor system. For instance, on one processor, in case of a problem, halting the system can be done instantly, whereas for a multi-processor, it will take some time.

3. significant communication delays may complicate matters, especially with geographically dispersed systems.

4. large systems are complex, and distributed systems tend to be large. Hence distributed systems tend to be complex.

They conclude that techniques and tools developed for sequential debugging are simply not adequate for distributed debugging. They also note that any debugging facility may interfere with the system, obscuring some bugs and revealing or even creating others.

They argue that:

- Debugging a distributed system should be done in a bottom-up manner, removing "sequential bugs" before going parallel, and thus leave only the hard bugs resulting from interaction between processes.

- Traces are essential, since observed results will often not be reproducible. Traces should be high-level. Still, traces may well be too large to be practical, and thus:
8.2. PARALLEL DEBUGGING PROBLEMS

- Debugging should consist of two phases: a first phase tracing and locating the error and a second where single processes are rerun with input taken from the trace, with the aid of a sequential debugger.

Trace data could be generated by the application programmer, but this process should be assisted by the system. The collected traces constitute a distributed database, and can be examined using standard tools. Sometimes intermediate states should be trapped using breakpoints, stopping all processes. One could then allow stepping, through statements or “significant events”. After initial debugging a monitoring system may be useful to “keep an eye on the system to make sure that things are running smoothly.” The master terminal should provide performance information and information about “important” events. They found trace files extremely useful, even without specialised postprocessing tools, and the monitoring facility not only for identifying abnormal behaviour, but also for demonstrating the system and for performing experiments.

LeBlanc and Mellor-Crummey [LCM87] point out that sequential debugging is usually cyclic\(^1\). This works because most sequential programs are deterministic, rendering the bugs reproducible. Since parallel programs are frequently not deterministic in general, and notoriously strange-behaved in the presence of a debugger consuming time and resources, the cyclic debugging approach does not work too well. McDowell and Helmhold [MH89] agree that cyclic debugging is unsuited for parallel debugging. They refer to authors claiming that attempts to gain more information about the program may contribute to the difficulty of reproducing behaviour, calling this phenomena the “Heisenberg uncertainty principle applied to software” and the “Probe effect”. In programs containing races, any extra statements will possibly alter the execution, and since debugging computation may well be costly, chances are that race effects are altered from normal runs.

Bemmerl [Bem86] points out that sequential debugging is difficult enough, but debugging real-time software for host/target environments adds difficulties:

- Debugging has to be real-time too, i.e. no slowing down can be allowed.
- There are usually no tools available on the target machine.
- Realworld processes are almost impossible to simulate, therefore it is difficult to generate test data, have to connect to realworld process.
- It is difficult to reproduce error situations.

McDowell and Helmhold [MH89] organise parallel debugging techniques into four groups:

**Traditional** debugging applied to parallel programs. This involves the setting of breakpoints on a per-process basis (traditional parallel debuggers). Breakpoints

\(^1\)The authors prefer the term *iterative debugging*, meaning exactly the same thing.
may disturb races, although this effect can be reduced by freezing local time on each node.

**Event-based debugging** generates and displays event-histories. These systems disturb the system as little as possible at run-time, and let the user go through the execution trace after completion, to see what actually happened. Since the trace is stored, the user may of course process parts of the execution any number of times and in his preferred order. This processing may be any (combination) of browsing, replay or simulation. Browsing lets the user see what happened. In the case of replay the program is executed again with constraints on what is allowed to happen at any non-deterministic point in the program according to the event history. The simulation approach lets the user run a traditional debugger on one process, giving external events to the process according to the event-history.

**Flow-and-control-display** to clarify the goings-on inside the system.

**Static analysis** of the program text, no execution required.

They observe that most current systems fall into the two classes “breakpoint debuggers” and “event-based debuggers”, although some combine features from the two classes. In addition, they found some static analysis tools.

Jones, Barkan and Wittie state and repeat, [JBW87, Wit89] that the two following problems are central to distributed debugging:

> “First, understanding the execution details of a parallel program running on many different machines is a formidable task. In general, too much data arises if the programmer tries to debug distributed systems using ordinary serial trace and dump information. Second, the precise sequence of process interactions that lead to an error situation in a distributed system can be complex, hard to isolate, and harder still to repeat.”

Peter Bates [Bat88] points out:

> “Extant interactive debugging tools rely on two important features of sequential programs: their time invariant execution; and the availability of controllable, accurate total system state. Neither of these can be reliably provided by distributed systems.”

> “Debugging distributed programs is a more complicated affair than that of sequential systems. In addition to the increased complexity of managing activity in distributed system, distributed system present difficulties for performing basic monitoring and experimentation activities. Timely access to distributed state, delivery of the selected state elements, experiments involving component synchronisation, and uncertainties about temporal relations among distributed components are among the more obvious difficulties to overcome.”
8.3. TIMING

It is important to realise that the problems of parallel debugging arise from several complicating facts, such as lack of synchronisation, several threads of execution, and multiple address-spaces. Other systems sharing some of these properties also share some of the problems. For instance, time-sharing systems may be unsynchronised and cooperating, with disjoint address spaces. Debugging such a system is very similar to debugging a parallel computer.

8.3 Timing

Chandy and Lamport [CL85] give an algorithm for detecting global states. This assumes that no global clock is available. On our hardware we can simulate a global clock by letting the host issue synchronising clock ticks, and in the system we can define the granularity of this clock to be good enough. Hence this article provides the solution to a problem that we do not have, at our level of granularity. Also, in the approach described in later chapters, we redefine time and concurrency so as to make life easier, see Section 13.7. To reformulate the general case, timing within one “time domain” (with a common clock) is trivial and exact [MY87], timing across different time domains can also be handled if the granularity of timing is coarser than the synchronisation of their clocks [BHL90].

Miller and Choi [MC88] discuss halting on breakpoints where no global time is available. They allow simple, disjunctive, conjunctive, linked and distributed predicates. The simple breakpoints are those which can be discovered within one node, the others require communication. They also give a halting algorithm based on distributed time. In short, a complex predicate is split up and distributed. When one part is satisfied, a halting discussion is initiated. This uses the concept of logical time from [Lam78], i.e. it is based on what possibilities different processors have to affect non-local computation, rather than physical time.
Chapter 9

Current Parallel Debuggers

9.1 Remote Debuggers

Remote debuggers execute on a different computer than the code being debugged, making them to some extent distributed. This makes them more difficult to implement than local, traditional debuggers, since the communication and interaction will be less reliable and flexible than the communication between processes within one computer. Remote debuggers do not really qualify as parallel debuggers as they do not treat a set of nodes as a whole, rather they are “sets of traditional debuggers”. For completeness, we would still like to mention a few things about them.

Peter Bates [Bat88] states:

“The ability to coordinate is the single important feature that separates a true distributed debugging tool from simply a debugging tool used on a distributed system.”

As an early distributed system, Grapevine [SBN84] had some support for remote debugging. Simple debugging could be done locally, but more complex corrections involving participation of experts had to be possible to do from the expert’s workplace somewhere else in the world. Therefore the servers included a way of getting access to the internal from the outside¹, to display statistics, alter parameters, and start jobs.

Redell [Red88] describes Topaz Teledebugging (TDD) which is a remote debugger. It can run multiple sessions against several remote hosts, but these sessions are still uncoordinated. This makes TDD a set of traditional debuggers rather than a real parallel debugger².

¹This was called a viticulturist’s entrance. A viticulturist is a cultivator of grapes, whose job it is to improve the Grapevine!

²The article discusses where to locate the TDD functionality on the target computer, concluding that it needs to go in to two places. One below all software, to debug the lowest layers of software called the mud (providing process scheduling, memory management and I/O drivers) and one higher in
Braner [Bra89a] tells us about tbug, the Trollius (see Section 4.5.3) debugger. Tbug can read in a symbol file and a node address, and print out the content of memory specified by address or logical name in various formats; this is all it can do.

In [Bør90] Børsting describes “A CHILL run-time system support for Teletesting”. He defines teletesting to be the same as remote debugging, where:

“.. teletesting [...] places a small debugger kernel within the program to be tested.”

This debugger is permanently available, since it is placed within the runtime support of a CHILL program. Remote debugging tools are called PXC tools (Program eXecution Control), and communicate with a general PXC component in the debugged program. This local component is executed as a special process, called a PXC daemon, and may pass commands on to be executed in the context of any specified process. This system is supposed to handle debugging of more than one node, but we have so far not seen any explicit mechanisms for controlling sets of nodes.

9.2 Breakpoint Debuggers

9.2.1 Nonparallel Developments

Olsson et al [OCHW91] describe Dalek, a sequential debugger. Dalek uses events, but primarily to get more advanced breakpoints, therefore we have chosen to see it as a breakpoint debugger. The authors find that a lot of current research goes into concurrent debugging, but that there is still a lot to be done for sequential debuggers, which may carry over to parallel debuggers later on.

Dalek has its own programming language, which can be loaded at any time or typed right in, and looks very much like C. This language is general, and has full access to the address-space of the program.

Dalek supports events, which can be raised conditionally at breakpoints in the program, or as dependents of other events. These events can invoke arbitrary Dalek code, which may again raise new events. High-level events may depend on sets of primary events, thus Dalek provides a dataflow up through dependent events. An event history records all event in an execution, this history may be used from the programming language or interactively by the programmer.

These facilities can for instance be used by a programmer to write code to “walk around” and verify the soundness of the current state, to be invoked only when an appropriate set of events are triggered.

the system. Experience showed that placing this too high, like a user process with access to all system services, introduced more problems that it solved, and it ended up just above the aforementioned “nub” and below the layers providing garbage collection, exception handling and dynamic thread management.
Dalek also has primitives for timing execution and access to the call stack, and can for instance be programmed to record time used in parts of the program, or number of calls to different procedures. The programmer can even tell the debugger to execute specific code as response to an event. This feature could for instance by a particularly adventurous programmer be used to first define the invocation of a function as an event, then bind the action of calling something else to that event, thus in effect dynamically patching code.

### 9.2.1.1 Evaluation

In fact, due to the power of the programming language, there seems to be virtually no limit to what the ingenious programmer can make the debugger do. To us, this approach seems to have only a very few possible drawbacks. We would expect it to be expensive to run, however, as the authors argue:

> "Although such high-level debugging might require a few minutes of computer time, it can save you hours of your time."

Computer time gets cheaper by the day anyway. The other possible problem is the degree of insight required by the user to be able to control it and use enough of the features to make it worthwhile. This is a problem that only experience can illuminate, and one that our PanParallacea may well share.

The event history allows the bugcatcher to see what really happened, this is probably necessary for debugging the debug script.

### 9.2.2 MPD

Ponamgi et al [PHK91] describe MPD, a multithread concurrent debugger, running on Mach on IBM's experimental 8CE multiprocessor. MPD is based on describing concurrent patterns which must or must not take place, through data-path expressions which are used for constructing a corresponding predecessor automaton\(^3\).

Data-path expressions specify causal dependence and independence among events in the same or different threads, and include a concurrency operator. They can not be used to specify timing constraints. They consist of an expression part and an action part, which describe a state and behaviour, respectively. Events can be things like invocation of a procedure or reading/writing of a variable. Events can be related by sequence, exclusive selection, repetition, and concurrency.

This system requires no modification to the source code. Breakpoints and actions can be added or modified at any time. It is implemented on top of gdb.

---

\(^3\)Predecessor automata appear to be an extension to finite-state automata, but their transitions are labeled with a program event and a set of immediate preceding events, allowing them also to detect partially ordered streams of events.
9.2.2.1 Evaluation

The system addresses problems with multiple threads within a process, but the current implementation is restricted to one process. We can not see that these ideas solve problems concerned with independent processes on one node or cooperating processes on different nodes.

9.2.3 Express

The Express system (Section 4.5.5) contains the debugger ndb, which offers a high-level dbx-like [dbx] interface, and a low-level adb-like interface [adb], see [ndb88]. This debugger uses the “set-of-processors” concept, in that each command is directed to a set of nodes. This debugger can halt computation on any processor accessing given memory locations, or attempting illegal instructions, and also produce per-processor core dumps. Processors can interactively be told to quit and dump core.

9.2.3.1 Evaluation

The “set-of-processors” concept here is used for giving commands. It must not be confused with our own “nodeset” concept, where a set of nodes is a logical entity in a programming language, being used for both command execution, expression evaluation, and logical operations.

9.3 Event-based Debuggers

Van Horn [Hor68] argues for design criteria for computers. His main point is that a set of criteria should be well founded, and used in all aspect of the design. He also argues that in order to facilitate debugging, three specific criteria should be met:

Input recordability:
All input to a program should be recordable.

Input specifiability:
In an execution, the user should have the possibility to completely specify the content of any input stream.

Asynchronous reproducibility of output:
Content and potential extent of the output streams of a user’s computation should depend only on the computation’s input streams and on the computation’s initial state.

Since the initial state and the content of the input streams are both controllable and input can be recorded, the user can run a program again and get exactly the same result. This is the underlying idea of event-logging and re-execution.
9.3. EVENT-BASED DEBUGGERS

9.3.1 Nonparallel Developments

This section gives some examples of very interesting work being done within sequential debugging, which up to now does not address any of the particular problems of parallelism, but which we hope will later be carried on in that direction.

9.3.1.1 PROVIDE

Moher [Moh88] describes a PROcess VIualization and DEbugging environment called PROVIDE. He says nothing about parallelism, but we will still mention some of its capabilities, to sketch the possibilities of event-based debugging. In PROVIDE, the main keywords are graphical presentation, deferred-binding animation and process history consistency maintenance. This system places great demands on resources, and anticipates advances in technology. Ideas that may at present seem less than realistic may well prove to be feasible later on, this is known to have happened within computing science before.

Graphical interfaces are considered to solve many of the human-computer interaction problems, and as such are important in themselves. Deferred binding means that the programmer can specify what she wants to see dynamically, at run-time. Windows can be created, where sets of values can be viewed graphically, on their own or combined. Debugging tends to be an iterative process, and this interaction is necessary for the programmer to be able to adapt to acquired knowledge about the execution.

The programmer works with virtual post-mortem dumps of program runs, at run-time, contradictory as it may sound, it is still true. The program is first interpreted, recording all state transitions in a database. All updates of variables generate new states. All debugging requests are relative to this database. If the programmer requests information about a state not reached, she has to wait until the interpreter catches up, in effect a pipe-lining a data-driven post-mortem inspection. A state can be specified in a number of ways, including statement counts and value criteria.

Evaluation The most revolutionary aspect of PROVIDE (and the reason for including it here) is the consistent maintenance of process history. If a value is changed by the programmer, the history from that point on is discarded and regenerated, based on the new value. The same is to be the case for alterations of code, when the programmer changes some piece of code, the system will determine the effects of this change, and regenerate process history accordingly.

Also, the flexibility in showing any expression graphically is an important strength, and one which will surely also be picked up by other systems.
9.3.1.2 Spyder

Agrawal et al [ADS91] present the Spyder system, which is based on execution backtracking. This system is similar to PROVIDE in some respects, it says nothing of parallelism, and it seems expensive to run.

The authors have analysed the debugging process, and found that it often works as follows: the bugcatcher looks at some variable, realises that its value is wrong, and wonders where its value came from. Intuitively, she backtracks. In practice, she sets a breakpoint at some earlier point in the program, runs to that breakpoint, and looks to see if things were as expected at this time. The breakpoint chosen need not be a good one, if it was too late, the process will have to be repeated. Also, if stepping over an important statement or not realising the importance of some statement, re-execution is necessary, and often costly in terms of time.

This process has three independent tasks: determining the statements which affects a variable at some point, selecting one or more of these for breakpoints, and restoring the program state at that point. Spyder assists in all these tasks, by first automatically finding where a variable can be updated, and then restoring earlier states of the execution.

Control statements (conditional and loops) alter the control location. Assignments alter the value of one or more variables. In both of these cases, the state prior to execution can be stored, and restored. Backtracking means resetting the state of an execution to an earlier state, this is achieved exactly by resetting the control location and variable values. The authors argue that not all states need be saved, if the programmer wants to backtrack into the $n$th iteration of a loop, the state prior to the loop can be installed, and $n$ iterations performed, this is in general true for composite statements. This can be used to achieve what-if analysis, by restoring a previous state, altering some of it and continuing execution to see what then happens.

Indirected assignments can be handled by storing both the value and the address where an update takes place. Input/output poses special problems, which can be solved by putting a layer over the I/O primitives and providing some kind of “put back” primitive.

**Evaluation** The notion of finding out not only that a variable has some value, but also where it came from, is very powerful. It is independent of parallelism, and would be very useful also in parallel debuggers.

---

4The system apparently also on the basis of other test cases assists in deciding which statements should be examined first. This sounds impressive, but it is not clear to me how it works.
9.3.2 TDCAda

A system for Testing and Debugging Concurrent ADA programs (TDCAda) is described by Tai, Carver and Obaid [TCO91, CT91]. The authors have addressed the problem that execution of a program with identical input may generate many execution paths, i.e. nondeterminism, with their deterministic execution. Their approach is to record information from one nondeterministic execution, and use this in subsequent runs, in effect rendering all parts of the program deterministic. They define the synchronisation sequence (SYN-sequence) of an execution to be an ordered trace of all sources of nondeterminism, i.e. communication between participants, timed events and access to shared variables.

The authors consider it impractical to collect all events of an execution. Therefore only enough events are stored so that the program can be executed again, with exactly the same execution path, i.e. what is needed is exactly the SYN-sequence.

When executing a program under the “guidance” of a SYN-sequence, the same sequence of events is regenerated, all nondeterminism is removed, and the work of the bugcatcher is substantially simplified, since she can reexecute the program and look at its states until the error is found. Also, since errors may occur only in some executions of a program, it is important to be able to reexecute an erroneous execution path after correcting the code, to ensure that the corrections were successful.

When debugging an Ada program $P$, two slightly different versions $P'$ and $P''$ are generated. $P$ is the real program to be debugged, and may be nondeterministic. $P'$ is also an Ada program, derived from $P$ through a mechanical process; it is equivalent but for small additions to save the SYN-sequence. $P''$ is a version of $P$ with some statements changed slightly and some control structures included. This program takes the same input as $P$ and a SYN-sequence generated from a run of $P'$, and reexecutes the path of $P$ (or really $P'$). This replay can be based on a partial or a total ordering of events.

The method described above is useful for finding bugs which occur in one specific execution of a program. The authors have extended the system to deal with systematic testing of nondeterministic programs, called deterministic execution testing. First a set of tests are generated, including input data and SYN-sequences. Then the feasibility of the combination of input and SYN-sequence is determined by trying to force an execution according to their specification. If the combination proves feasible, the results are examined.

The transformation for debugging is similar to that for testing, but somewhat simpler. The debugging version can assume that a SYN-sequence is valid, and needs less information.
9.3.2.1 Evaluation

In this way, even a nondeterministic program can be thoroughly and systematically tested, but it is not clear how large the SYN-sequences will be in practice; if they are too large the practicality of the method will suffer.

This language-based approach to debugging has the advantage that it is as portable as the program, and completely independent of details like the implementation of the language, operating system, and hardware. It is also easier to develop language-based than implementation-based tools, but they are usually less efficient. Developing language-based solutions will often lead to better implementation-based solutions, and may as such be worthwhile. Unfortunately the authors had to make some assumptions about the use of the language (such as the use of uninitialised variables), but these seem to be technical, and pose no serious problem.

The main virtue of this system is that the non-determinism problem is solved. Through the test generation method even bugs in some executions hidden due to a particular execution sequence are brought into the light, and the intrusion problem is thus also solved!

9.3.3 Eden

The execution monitor Eden [AFC91] is developed at Kyushu University, and seems to have nothing to do with the object-oriented operating system Eden, apart from the name.

As TDCAda, Eden transforms Ada programs. A program $\alpha$ is transformed into a $\alpha'$, with entry calls to a runtime monitor inserted. These calls are used to store an execution trace, to be interactively inspected, or used for replay. Eden detects deadlocks by updating “task-wait-for” graphs.

9.3.3.1 Evaluation

The authors argue that tracing is important in a nondeterministic system due to lack of reproducability, but say nothing of the invasion by this system, and how that affects races.

9.3.4 PPUTT

The toolkit of the “Parallel Program Understanding Tools and Techniques” (PPUTT) group at the University of Rochester, described by Fowler, LeBlanc and Mellor-Crummey, [FLM89] records a history of access to shared objects per process, which seen together represent an execution history. This can then be represented graphically, and zoomed in and out, providing an abstract view of an entire execution. This
execution history is used for replays. In the graphical tool, each event is displayed as a box, where the length of the box shows the time it took and lines between boxes shows dependencies. They see their tool as a: "valuable resource for debugging and performance analysis." It tries to keep the following properties of sequential debuggers: analysis should be top-down, repeatable, interactive, potentially very fine-grain, and extensible.

### 9.3.4.1 Evaluation

Replay is covered, but intrusion is not. The graphical interface seems nice.

### 9.3.5 Behavioral Abstraction

**Behavioral Abstraction** (BA) described by Bates and Wileden [BW83] offers a new approach:

"The basis for behavioral abstraction is viewing a system's activity, or behavior, as a stream of event occurences."

Primitive events are basic actions in the system, e.g. process creation, message transmittance, etc. They can be clustered and filtered to represent higher-level events. The paper discusses how to detect complex events, and how to ignore "noise". The authors consider time too difficult to handle, even with global time algorithms the granularity will be too coarse. Events can be used for presentation to increase system understanding. An event can also be bound to a specified action, as normal breakpoints.

#### 9.3.5.1 Evaluation

LeBlanc and Mollor-Crummey [LMC87] see two disadvantages to BA systems: the user must before execution specify all interesting events, since further probing after discovering the error is not possible, and the information storage requirements quickly make the method infeasible. Also, the intrusion must be significant.

### 9.3.6 EBBA

The **Event Based Behavioral Abstraction** (EBBA) [Bat88] is connected with BA in the previous section. EBBA is a set of tools which together help the user manage complexity in a heterogeneous environment. The user builds up a model in terms of events, and matches it with actual system behaviour. Event can be primitive or high-level. Then this model may be changed to investigate the system from some other perspective.
The authors sketch a set of different ways the tools can be used:

**Simple remote debugging:**
Remote nodes recognise local events and sends them to a central node where the bugcatcher can use the higher level tools.

**Primitive event collection and posting:**
Here, the events are distributed to suitable cooperating nodes in the system.

**Filtered remote debugging/preset actions:**
Performance can be improved if the local nodes keep an *attention list* over events they need to inform about, and possibly actions.

**Simple cooperative debugging:**
If all nodes are allowed to examine the network event stream, less explicit communication is needed.

**Distributed debugging with distributed event recognition:**
Model abstraction is done on each node, and higher-level information is communicated.

**Centrally directed:**
Now, the model builder resides on one node, and the librarian collecting events becomes accessible to all high-level recognition nodes, as an event model server.

### 9.3.6.1 Evaluation

EBBA seems to address some of the problems of BA mentioned above, but the practicality of these systems are less discussed than the theoretical properties, and it not clear to us how easy they are to use. This system must be intrusive, but surely events can be used for replay?

### 9.3.7 Instant Replay

One solution suggested by LeBlanc and Mellor-Crummey [LC87] is **Instant Replay**, providing repeatable execution of highly parallel programs in tightly coupled systems, extending to loosely coupled systems. Their solution consists of saving the relative order of significant events as they occur, not the data associated with the events. They assume that processes do not contain nondeterministic statements, and do not allow asynchronous interrupts. All interactions between processes are modelled as operations on shared objects, a series of modifications to a shared object is represented as a totally ordered sequence of versions. There is a "history tape" identifying the sequence in which processes update the object, this is used to reproduce the sequence of total states. When replaying, only the same update sequence to each object is allowed. Breakpoints during replay will thus not alter the execution path,
and even program alterations are allowed, as long as they do not affect the sequence of interactions to shared objects from each process.

9.3.7.1 Evaluation

This system gives the programmer a replay facility. Due to the assumption that the data of events are as deterministic as are their order, data storing is not used. This reduces the storage demands and time consumption significantly.

Unfortunately, the determinism assumptions taken seem rather less than universally valid, thus reducing the applicability of this approach.

9.3.8 Bugnet

Bugnet [JBW87, LBPM88] is a real time distributed debugging system. It uses timestamps for logging and replaying events, based on checkpoints [WC85], instead of logical time and partial orderings as chosen by other systems\(^5\).

Bugnet allows the programmer to execute code and roll back to previous states, thus allowing her to find out where an error comes from. During replay, the user can choose which processes to reexecute. All communication must use messages, and all I/O and communication must use system primitives, otherwise their timing and effects can not be logged. During replay the individual processes can be:

- **dead**: not executed, messages from the original process are generated from the log.
- **non-sending**: executing, but messages are not sent, rather generated from the log.
- **sending**: executing and sending, original message must be filtered from the log in order not to occur twice.

The individual process can be executed continuously or via single steps. In a typical session, all processes being debugged would be non-sending, but the generality of the system gives the user very interesting possibilities.

9.3.8.1 Evaluation

In a summary from a presentation of Bugnet, [LMC87], the point was raised that since communication primitives in this system are synchronous, it is not clear if there would be any difference between logical time ordering and the scheme used. Also, it was argued that the logging system, since it reportedly slows interprocess communication down by a factor of three, would mask time-dependent errors.

\(^5\)The reason for this is that logical time does not handle node-local events such as interrupts in an acceptable way. In this scheme, local clocks are synchronised frequently, and considered correct between corrections.
The creators of this system consider timing so crucial that the difference between logical time and local time is important. It seems to us that a logging system of such cost would affect the execution heavily, thus by their own argument leaving the system of little usability?

The re-execution of only some processes whilst simulating the others is an elegant one, it allows the programmer to focus in on one part of the program, and also reduces the cost of re-execution.

9.3.9 DISDEB

Lazzerini and Prete [LP86] describe their “Distributed Interactive Symbolic DEBug- ger” (DISDEB). This debugger runs unmodified code, and apparently only requires minor modifications to the OS kernel. Instead it makes use something rather elegantly termed “ad hoc hardware.” This essentially hardware-based approach leaves the processor running at full speed, hence races are not affected.

The system is designed to debug both system and applicative software. It consists of a high-level command interpreter and software modules executing on dedicated “Programming Debugging Aid” (PDA) boards. The complex commands of DISDEB are translated into sets of elementary commands, to be executed on the PDAs. Results are processed before being passed on to the controller.

An elementary event for memory locations consists of a specification of: process identity, address, width (byte or word), access type (read or write), a relational operator and domain/range of the access. A similar definition exists for input/output channels. An event can be either: an elementary event, a sequential conjunction of events, in instantaneous conjunction of elementary events\(^6\), or the logical disjunction of events. Events can be local or distributed. On events the following actions can be taken:

**Define:** binds an event to an identifier,

**Activate:** i.e. turn insertion into the event list “on”,

**Deactivate:** the opposite of activate,

**Break:** to stop execution of specified processes,

**Trace:** to keep information about all modifications to a specified memory location,

**Start.timer:** to start a timer whose value is presumably used somewhere else in the event response program,

**Stop.timer:** stops the timer,

**Start.counter:** sets a specified counter to zero, this counter may be used elsewhere

---

\(^6\) Note that this instantaneous conjunction is restricted to elementary events, unlike the rest?
9.4. PARASIGHT

Increment_counter: add one to a counter, and

If: a normal conditional statement.

9.3.9.1 Evaluation

Lazzerini and Prete [Ibidem] do not comment on reproducibility, timing problems are solved by the PADs. Without having used the system, we are a bit surprised that the actions include a conditional statement, but no iterative statement.

DISDEB recognises sequences of events. This is similar to what we have done, we discuss that in Section 14.4.

9.4 Parasight

Aral and Gertner [AG89] present Parasight, a high-level debugging tool. Since we have some problems deciding on what category of debugger it is, we have given it a section of its own. The name seems to have to do with the use of parasites in the code. The underlying idea of the system is to use the power of parallel processing to reduce intrusion.

Parasight uses scan points, which are "light-weight instrumentation primitives", inserted and deleted at run-time. Scan-points are implemented by patching branches into the executable code. The patch is a jump to general user-specified code, with general power. If the called code is efficient, the code of such a branch is minimal. This code would typically only store status.

In addition, a set of parasites execute concurrently, typically on their own processors, also executing user-specified code. Since this code runs on several processors, it will not interfere with the application code except for it using some of the processors. It can then monitor data-areas used by the scan-points, and thus react to events in the program. This parasites use general, user-defined code, and can be used for "anything programmable", like timing, profiling or debugging.

9.4.1 Evaluation

The intrusion in the system comes from the scan-points, and can be made ignorable. The system seems to use the facilities of shared-memory multiprocessors in a very clever way. In a multicomputer implementation the scan-points would need to transfer information over to parasite nodes, thus increasing intrusion. Also, the parasites possibility of inspecting or modifying data structures would be limited.
9.5 Others

Trew and Wilson [Te91] in their description of parallel computers and companies also mention several debuggers. We mention some of these and some others here to give an outline of what programmers can realistically expect to get to work with.

The Connection Machine has an extended dbx, with facilities for examining parallel data structures. It also offers gprof.

The FPS T-series\(^7\) is a T414 (Transputer) based MIMD computer, using one Weitek chip per node for processing power. It comes with parallel versions of dbx for symbolic and assembly language debugging, and parallel gprof.

The BBN, a shared memory, medium grained MIMD computer offers TotalView for debugging:

"TotalView is a menu-based debugging system. [..] It is possible to set conditional and unconditional breakpoints and evaluation points within each process interactively. The CPU overhead is estimated to be less than 5%."\(^7\)

Chorus Systèmes (suppliers of the operating system Chorus, see Section 4.5.1) offer three different debuggers: KDB, GDB multi-thread and CDB. Their KDB is a kernel debugger, allowing the bugcatcher to freeze and inspect the entire kernel. The GDB multi-thread is a version of the GNU debugger GDB modified for thread handling, in effect halting all threads when one thread encounters a breakpoint, and allowing inspection of local variables and stack of other threads than the halting one. CDB focuses on communication, through displaying information on ports, port groups, and message queues. We have no documentation on these debuggers, but Chorus representatives indicate that KDB and GDB are extensively used, CDB less so as it is less intuitive or familiar to users.

\(^7\)FPS Computing was previously called Floating Point Systems Inc.
Chapter 10

Related Tools

In this chapter we mention two other programming tools, logic analyzers and simulators. Apart from these, the programmer also often has interpreters, memory dump utilities, audit and trace packages at her disposal.

10.1 Logic Analysers

The logic analyser is primarily a tool for low-level debugging. It can be connected to various points on boards, between a processor and the socket on a board, and in some cases it can connect directly into special debug editions of processors. The analyser can be programmed to "trig" on specified conditions, including logical expression specified over the information available on the input sources, and time dependent transitions.

The author has no personal experience with this tool, but it is heavily used and highly praised within the group developing the HC-prototypes. It is the only debugger available before customised debugging can be implemented. Sometimes the user interface can be a bit complicated; once understood the effort is well worth while as it opens for an abundance of information.

10.2 Simulators

Computer simulation can be many different things. A computer can simulate the behavior of real-life systems, to investigate the underlying properties of the real-life system.

Computers can also simulate other computers. Emulation is one special case of this, where a program written for one system can be run on top of a different system emulating the behaviour of the other. They can also be used to simulate hardware not yet realised, not yet bought, or not available to the user.
A software simulation system can often be run on standard hardware; the efficiency may be low but the accessibility very good. A company owning just one large parallel computer, will often have to restrict the usage of it. Using this resource for repetitive debugging runs is often out of the question, but a simulator run on the users workstation can be used for experiments without affecting other users.

A simulated system can often give functionality not implemented by the "real" system. The simulator can thus be used to evaluate the applicability of algorithms and implementations before investing heavily. Simulators of parallel computers run on one computer can make it easier to debug parallel programs. In good implementations, fairly advanced instrumentation can be performed with greater ease in a simulated environment. On a "real" system, the user will be restricted to the actual behaviour of the system, a simulated system may offer possibilities for behaving in different ways. For example, the user may have the option of simulating her program with relatively higher or lower cost of individual operations like inter-node communication, thus revealing algorithmic properties otherwise hidden.

The author spent six months at Philips Research Laboratories during the winter of 1989/90, implementing a system simulating the execution of programs written in POOL, executed on their prototype computer POOMA or similar architectures. This work is described in [Moe90b], included here as Appendix B, also including references on both the language and the prototype computer.

The “Portable Pool Simulation system” (PPS) is integrated into a portable implementation of POOL-X. When simulating, a system can choose to estimate time for some operations and measure actual execution time of others. This can be relevant, if the target system and the workstation are similar, but this can be a dangerous assumption to make. All execution times in PPS are estimated, and do not limit the portability of the system\footnote{The portable POOL implementation and PPS are programmed in C and run under UNIX, and can as far as the author knows run on any computer providing reasonable versions of these.}. There is a file of constants which a user may (but need not) modify to simulate a computer behaving according to her personal estimates/tastes/needs. The system is even prepared for self-modification; the maintainer can run a set of programs to find the actual values for the timing parameters. This is useful as the system evolves, and the performance of the individual operations is changed.

The user can run this simulator on any Unix workstation, and she can simulate any behaviour she wants to. The system also gives the experienced debugger a unique insight into what really goes on in the computer. As we see, it provides portability, accessibility, and a way of evaluating architectural changes.
Part III

$HC - 2^d$ and debugging
In the first chapter in this part we describe the Hypercube project. We carry on by discussing what support we would like the programming environment to give the programmer, both with respect to monitoring and debugging. Then follows the description of a novel debug control language, and a few examples hopefully indicating its strength and flexibility. Briefly, we have sketched how an interactive interface to the same system could look. Then we describe how PanParallacea has been implemented, thereby clarifying many details about how the system is meant to work. We conclude this part, and with it the main body of the thesis, by enumerating the contributions in the thesis, the status of the implementation of the system, its weaknesses, and directions for further research.
Chapter 11

Instrumentation and Debugging on HC16

This chapter will describe the projects within which we have done the research described in this thesis. This current Hypercube project is a followup project to the CROSS8 project, both headed by Professor Kjell Bratbergsengen, and both have been supported by the Norwegian Government Research Agency (NTNF). Their development is described by Bratbergsengen [Bra89b].

Tools developed are intended to be used primarily on the computers developed within this group, which are Intel-x86 based parallel computers, connected through neighbour-shared memory in a hypercubic topology. Assumptions made will to a varying extent depend on the characteristics of these computers. The interconnection topology is not important. It is important for the implementation that the host has an independent communication channel to each node, but not for the design. Several aspects of the operating system and the communication system are crucial for the implementation and also for parts of the design, for instance it is necessary that communication can take place without scheduling, and that threads are not preempted. If this was to be the case, several alterations right through the design of the system would be called for.

11.1 Terminology

For the rest of the thesis, we will repeatedly use a set of terms which may be unknown to the reader, or used in a slightly unusual meaning. We would like to define some of these. We do not advocate our use of these as a standard, in fact some will definitely not be useful in general, however, for this text they seem appropriate.

Host program or host interface is a program running on the host computer. This program controls all communication between the hypercube and the outside world. Messages from the nodes must be forwarded from the host program in

135
order to reach the debug controller. Our current host program also provides a
user interface for loading programs\(^1\), and is shown if Figure 11.1.

**Debug controller** is a program running on a workstation, controlling the execution
of the debugger and interacting with the user. In our case we have a single *d.c.*
controlling more than one node, giving the user an integrated view of the nodes.

**Debugger** is the collective name of the sum of the components implementing a de-
bugger, as we will shortly describe that includes program executing on the target
computer, the host and on the debug controller workstation.

## 11.2 Projects and Prototypes

These projects are successors of several previous efforts within database research in
Trondheim. The main aim of our projects has been to look into methods and algo-
rithms used in database systems on multicomputer systems. We have also looked into
some related fields, e.g. image processing, ray-tracing, numeric operations, parallel
sorting, and parallel programming environments.

Each node of these computers has its own memory and its own disk. Database re-
lations are split into fragments, with one part stored on the local disk of each node.
During database operations, the tuples of relations may be redistributed, based on a
hashing scheme. The project and some of the methods are described in [Bra89b].

### 11.2.1 CROSS8

The CROSS8 project was started in 1985, its purpose was to examine the evaluation
of relational algebra operations on parallel computers.

The project designed and built its first prototype computer, also called CROSS8, in
eighteen months. This prototype was running the *Wisconsin Benchmark* [BDT83], by
some authors called the *DeWitt join test*, in March 1987. The prototype computer
consisted of eight nodes completely connected over pairwise shared dual port memory
chips (DPRAM). The nodes were based on the Intel 80186 processor, and had one
SCSI disk each. The DPRAM connection between each pair of nodes was one kilobyte.

CROSS8 was connected directly to a personal computer running MS-DOS, and pro-
grammers had to sit physically at this personal computer (host) to use the computer.
The software adhered to DOS standards, and could be compiled and linked at the
programmers own personal computer.

The CROSS8 project was gradually transformed into the the Hypercube (HC) project.

---

\(^1\)This function might well be separated out into an application program using the communication
offered by the host program.
11.2. PROJECTS AND PROTOTYPES

11.2.2 HC16-186

The work on our first HC prototype started in the spring of 1987, this computer was finished in May 1988. It was based on a slightly quicker version of the same Intel 80186 processor, but had more memory and larger disks with integrated controllers. Most importantly, it had full bus width to the inter-node DPRAM, which was still only of two kilobytes. It was called the HC16-186, where 16 is the number of nodes and 186 refers to the processor model. For this and larger number of processors it was decided that the all-to-all crossbar connection scheme was too complex and too expensive to implement, and also had more performance potential than the nodes could utilise. The hypercubic topology [BLRA80] was chosen instead, and has since been used for all subsequent designs and prototypes. That is the reason for the letters HC in the name. This computer was fully controlled by software, including the stopping and starting of disks. This was an important step towards making the system available over the network.

When finished, HC16-186 promptly proved to perform better on relational algebra operations than any other computer [Bra87]. A comparison with other similar computers [AT90] shows that the communication of this prototype had significantly lower latency and higher bandwidth than competitors based on much stronger processors and having more specialised hardware.

11.2.3 HC16-386

After the success with the implementation of HC16-186, the design and implementation of HC16-386 was done in about a year, this prototype was finished in December 1989. As the name suggests, it is connected in a hypercubic network and has sixteen nodes based on the Intel 80386 processor. This computer is similar to HC16-186, and due to experiences with that computer the -386 version was a much more mature design, and also significantly more stable.

11.2.4 HC64-486

During 1991 work continued towards a multicomputer based on the Intel 80486, with 64 nodes. Several improvements go into this design, both due to experience and a stronger processor with more functionality. We have also opened up for adding vector processing capacity on each node, to make it a direct competitor to other supercomputers. This is a current research area in the group.
11.3 TorOS

Each single node runs one instance of an in-house developed operating system, called TorOS, described by Brekke [Bre90]. The operating system has the same call interface as Unix, but with several modifications to support events. The message passing is handled outside the operating system, this is described in more detail below.

11.4 Monitor

During the autumn of 1990 Kåre Nævdal and Vegard Stenvold, two diploma students at the department, undertook the task of designing and implementing a run-time monitor for HC16-386 as part of their joint Masters Thesis work. This was appropriately enough called Monitor². Their work was made difficult by several facts, for instance that they started it before anyone really knew exactly what they were supposed to be making, and that the hardware and software were under continous development at the same time. Nevertheless, they implemented parts of a system ranging from operating system components to an X interface. Their system worked well, showed what was happening on the computer, and provided an excellent starting-point for the implementation of the integrated monitor and debugger described in this thesis: PanParallacea. Their effort is described in [NS90].

The monitor system has influenced the operating system. Events interesting for monitoring update counts when they are executed. This is done inside the operating system, and is always present. Once every second a message is constructed, and sent out to the host program. The contents of this message is designed to be modifiable at run-time, by sending messages in to the monitor thread. This was however considered an important feature and never implemented. The system now always sends full messages out, and the overhead is still hardly noticable.

In the host program, if there is a debug controller connected, the messages are forwarded over TCP/IP to it, otherwise they are just discarded. This continous sending of messages increases the chances that the system performance will not be altered by monitoring, since the action on the nodes is always the same.

In the debug controller, a set of variables portraying the state of the computer is updated, and if so requested the controller saves the message for later replay. The display is updated periodically, and the display is independent of the monitor kernel, in that the graphics are generated in the debug controller.

In the requirements definition of this system, we find that information is split into five groups: CPU-usage, communication, I/O, memory usage, and interrupts. The computer is to be monitored on different levels:

²This refers to overseeing a situation, but oversees a computer, not a data-structure like Hoare’s monitor which we mentioned in Section 4.1.1, with which our monitors have no relation.
Node-level:
This level shows all nodes at the same time, although the name may lead one
to think otherwise. This is primarily useful for detecting a skewed resource
consumption. The user can get information from all the classes mentioned above.

Job-level:
A job is a collection of threads, operating in the same address-space, see Sec-
tion 1.2. On this level, essentially the same information as on the level above is
accessible, but specified per job.

Process-level:
As the alert reader may anticipate, this is a level where resource consumption
is specified per process. Memory usage is not relevant on this level, as memory
is allocated per job.

Procedural level:
This level informs about the invocations and returns from the individual proce-
dures of a program.

Event-level:
The authors suggest collecting information about events, possibly limited by
evaluating some criteria.

The implementation of this system was unfortunately not quite completed.

Several observations were made. The authors estimate the cost of monitoring to be
at most 0.6 %, since 6 ms is the time consumed by the monitor thread awakened once
a second. This conclusion seems a bit simplified, as it does not include added cost of
performing monitored system functions, and overhead by the communication system
when sending messages out to the host. It is also observed that less frequent polling
will reduce the cost of monitoring, but also its accuracy.

One problem with monitor is the granularity. The sampling period was one second,
and this could not be changed. For some programs, this is far too coarse, as all
execution finished in less than a second. Scaling up the problem is not always a
solution, since that will sometimes change the implementation. In one case we wanted
to monitor a program looking through a small database once. This small database
fit in main memory. If we were to scale it up, it would require the use of disk-based
methods. As we see, scaling the problem would change it too much to be useful.

The first version of the instrumentation environment worked well. Programmers
gained useful insight from it, which led to improved programs. The need for a de-
bugger was still very present, the debugging methods of choice on HC16 are printf
statements and the logic analyser for the more determined souls.
Figure 11.1: The host program xhc
Chapter 12

Instrumentation Requirements

This chapter discusses what support we think the programmer of HC16 needs, what information the instrumentation unit should provide in its different modi, and how it should behave. Debugging needs are described in the next chapter. Where considerations are common to instrumentation and debugging, we have tried to put them in this chapter.

12.1 User Interface

We have to agree with Bemmerl [Bem86]:

"During debugging, the programmer should be able to use the same abstraction concepts as when writing his programs."

We find this relevant for both monitoring and debugging. The debugging process is difficult enough as it is, and it is important that both monitoring and debugging are presented in terms which are easy to comprehend by the user. Monitoring plots should be presented graphically, preferably real-time with an option of generating paper copies. When the programmer gets information from multiple sources, e.g. program profiles, monitor plots, and debug information, it must be easy to see how the information is related.

An interactive tool for monitoring parallel computers might be better designed as a family of separate units, as we have seen several authors do in previous chapters. From a software engineering point of view the choice of how to organise a set of tools with much but granularity in common is surely important; in our case, we want the programmer to combine instrumentation and debugging in such a way that information connected with both activities must be integrated or at least available together. For now we have chosen to let the issue of organisation drown among others as all tools are clashed together.
In some systems, the user has to decide in advance if she wants to be able to debug or profile the system. There are good reasons for this, primarily easier implementation, cleaner systems, and less run-time penalties. Unfortunately, the need for "system inspection" is not always known in advance. As Redell [Red88] convincingly argues:

"A user who has just been thwarted while closing in on a network server bug which strikes after 1000 hours is unlikely to be placated by the motto "better luck next time"."

We would therefore like the support system to be available at all times. It should however be unnoticeable until called into action and not slow down the normal operation of the system. A debug process can lurk in the system with low priority at all times, awaiting the call for its services, like in the Chill/Chipsy system [Bor90], or it can be started when needed.

Also, this should give the user the choice of altering the level of system surveillance at runtime, e.g. invoking the detailed monitoring or even the debugger when noticing from the rough measurements mentioned above that it might be interesting to inspect the execution more thoroughly.

It must be possible to start instrumentation unit and debugger from the host-client, by pushing a button.

Numeric information can be displayed in a number of ways, all surely having their advantages and advocates. In our system, instrumentation data must also be available numerically to the debugging script process\(^1\). Initially we will aim for two forms of graphical presentation, one for showing instant values and to show how values change over time, but we will be open to other alternatives and extensions in later versions. Both presentations should be available for all types of data being displayed, and possibly at the same time. We want to see columns of monitorable values prefixed by two buttons, one for instant value displays, and one for values over time. We prefer slide bars and scalable histograms for presenting these values, respectively.

Performance numbers from simulation or instrumentation will have to be smoothed, otherwise they often end up with a series of peaks and valleys, quite impossible for humans to understand. In our case, we will use smoothing for the values over time, and no smoothing for instant value displays.

Too much detailed information will give no overview, it has to be summarised. This also limits the exactness of the information, ideally we would like to "zoom in" on parts of a smoothed display to see the finer details.

\(^1\) This will be explained later, for now it suffices to note that instrumentation data will be used for other purposes than mere "flashing of lights."
12.2 Monitoring

Debugging and performance debugging tend to be iterative processes. In the different stages of these processes, the needs of the programmer will vary. Therefore, instrumentation should be available on different levels. Also, the need for exact control will vary from non-existent in the case of “toy programs” to dire in the case of real-time and/or production-quality programs. In the case of some commercial, parallel systems it seems likely that more effort may go into optimising the package than what originally went into implementing it. The monitoring tools must support a wide range of granularities.

The programmer will need not one but a family of supportive tools to obtain insight to the exact extent that she wants and needs at different times. From the UNIX world we already know such indispensable friends as for instance `prof` and `tcov` which we mentioned in Section 6.2. At some point in the development cycle, these can be extremely helpful to any kind of project. We also know the performance meter style of tools, like `perfmon` (also mentioned earlier), which enable the user to observe at runtime how a system is behaving. Even within the class of run-time monitoring systems there is a large gap between minimal, single-metric tools such as `istat`\(^2\), and multi-metric tools such as for instance `perfmon`. We have included an example output from these programs in Appendix A for readers not familiar with this kind of tools. The foci of these are very different, but as we have argued before (in Section 6.2) their use is sometimes overlapping. Profiling tools allows the programmer to examine exactly where the time is spent in the program, with references to function names and line numbers, and thence to focus her ameliorating\(^3\) efforts accordingly. Performance meters give the user the state of the computer, real-time and without reference to the source code; a knowledgeable programmer will often be able to deduce from this information how to make the program behave better. The monitoring toolcase should include both profiling tools and pure monitors.

In order to improve efficiency efficiently, the programmer must know not only where cycles are wasted, but also how much the waste amounts to. Ameliorating a minor, only sporadically used function will not improve the situation much; it will often be a waste of time and effort. Therefore the system must provide a list of resource consumers sorted on consumption. As the rest of the system this should be flexible enough to provide information per job, thread, and function. This is a part of what profiling is well suited for, and need not be presented interactively.

We have chosen to focus our effort, at least for the time being, on interactive and debug-related aspects, and thus leave the topic of profiling for now. Do however not take this decision as an attempt to reduce the importance of profiling, the authors consider it essential!

\(^2\) `Istat` is an X program, which in one window show a histogram over load average, and also informs about disk transfer rate, network I/O rate, and CPU usage distribution(system, nice and user CPU time). `Istat` typically sits quietly in the corner of the screen, waiting patiently for the careless glance of the user checking if the computer is still alive.

\(^3\) Optimising
We would like our interactive monitoring assistant to cover the range from cheap insight to exact knowledge, with debugging facilities integrated into it as well. Taking it for granted that increasing the level of detail will imply increasing the invasiveness of the debugger ([MH89] and others) and thus the distortion of the performance, we realize that a problem is at hand.

In order to avoid invasion into the execution, monitoring can be done by augmenting hardware as suggested by Plattner [Pla84] and others. It seems from superficial contact with literature regarding hardware assistance of monitoring and debugging that some problems can be solved, but that integrating the added hardware components and the rest of the system is one complicating factor. Two basic approaches are obvious: hardware sensors can detect conditions without invasion, and activate software as appropriate. Added processors can listen in on the computation, and collect information on a separate processor. Researchers working on solutions involving hardware assistance seem to have some way to go before producing the ideal system, and we trust that this reflects some complexity in the task. Due to my lacking knowledge of hardware construction and to limit the scope of this work, we have chosen not to investigate hardware support or implementation.

Software monitoring will always affect execution. Our system is definitely not deterministic, and monitoring and debugging are bound to distort the execution of programs. We have chosen to live with this. When only monitoring and not debugging is in use, the amount of work spent on monitoring should be similar on all nodes and evenly distributed, but sometimes even small differences will matter. We choose to say that a system should be stable so that a little added computation involves only proportional distortion. Then a monitored system should be so close to an unmonitored system that the changes motivated by monitor observations are also relevant and helpful when the program is run without monitoring. Since we have chosen to have the option of instrumentation available at all times, part of the overhead is present both when monitoring and when not. This costs some CPU cycles, but makes the system easier to use, and as we see in a circumstantial fashion reduces distorting intrusion. In the case of debugging, execution will be heavily distorted. The programmer must face this by writing programs that do not depend critically on timing. "Besides, it builds character."

These tools should give the possibility of pinpointing what resource constitutes a bottleneck at any time. As argued by Bennert [BHL90] the latency of a service in combination with the load can tell if it is a bottleneck. If the load and latency are both low, the resource is not utilised. If the load is high and the latency low, the resource is in active use and works well. If the load is low and the latency high, the resource may constitute a bottleneck. If both are high, it is definitely a bottleneck, and the programmer might consider reducing or parallelising the use of the resource. We must therefore look into the possibility of seeing delays in a service in conjunction with the utilisation of it. The delay information should be presented as instant values and over time, as all other parameters of the system.

Comment [CG90], referring to the novel consequences of work needed to convert Linda inspired programs to something runnable on non-Linda systems.
We have tried to design the system in such a way that adding a review mode at a later time should be possible; if this is successful we will only know when and if we ever try. A review mode would be useful for looking at stored trace data, especially for exact performance debugging. The interactive mode will give an overview, but of less generality because the numbers displayed may show complex correlations rather difficult for the programmer to comprehend at runtime and at "real-time" speed. The real-time and the review unit may be constructed as one program or two different programs. This issue is postponed.

For this review mode we need to store traces during execution. This tracing may save all information on node, a mirror-node or on the controller workstation. If stored on the nodes, information may be shipped across as needed, or before starting review. The first approach is more elegant and probably efficient, the latter probably simpler. The first implementation of monitor saved traces on the workstation, this decision should be reevaluated in later versions.

12.3 Monitored Parameters

As we have sketched how the system should look and feel, we now turn to the question of what parameters the programmer will need to monitor. All information must be presented on various levels, including per thread, per job, and per node. In addition to the parameters mentioned below we would like to see statistics about consumption of things like memory and bus bandwidth, but that would probably require hardware assistance not available on our system, and is thus excluded from the list below.

12.3.1 Processes

The monitor must give the user a list of the jobs running at the nodes at any time, and information about the threads in each job. This should include thread names, status, and a summary of time used. To get a deeper insight into execution potential the programmer will also need to see the number of threads ready for execution, those waiting (for I/O or communication), and those threads existing but inactive (sleeping). It should also show distribution of delays from entering each queue to coming out on the front of it, scheduling frequency and duration of executions.

12.3.2 Processor Usage

The processor usage is a very important metric in any CPU bound system, and is traditionally split into usage for user processing, system work, and idle time. These parameters can be shown as percentages, adding up to one hundred.
12.3.3 Communication

In a MIMD computer, and especially one programmed in an explicitly parallel way, communication is of major importance, but the parameters will be system dependent. As described in Section 18.3.1, the system is activated by interrupt when a message arrives, if the communication load is not too high. This number of interrupts must be displayed. Also, the number of messages in the message heap not yet requested for reception by local threads is of interest. In addition the more usual numbers of both attempted and successful sends and receives must be shown, together with the delays in message handling.

12.3.4 Memory

The memory usage of the jobs is interesting for several reasons. In systems with swapping or paging, high memory usage may lead to thrashing, and reduced performance. In systems without swapping or paging, excessive memory usage may limit the number of coexisting processes on a node. Knowledge of the memory usage will also be useful for both hardware and software designers, in their building up information about system usage.

12.3.5 Disk

The disk usage in this system is similar to other systems; we need no special features for monitoring disk usage. We have found it useful to see four metrics concerning disk usage:

<table>
<thead>
<tr>
<th></th>
<th>Attempted</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>Read calls</td>
<td>Reads from disk, not including cache hits</td>
</tr>
<tr>
<td>Write</td>
<td>Write calls</td>
<td>Cache flushes and synchronous writes</td>
</tr>
</tbody>
</table>

The system should also inform about average and distribution of delays for the various calls. In a system with paging or swapping, the disk usage statistics should specify how much of the traffic is related to that.

12.3.6 DBMS

When the computer is used as a database server, the programmer will also need to know about a set of application specific parameters. This section is not central to the thesis, the application is also not at a stage where the set of interesting metrics are clear, they will depend on implementation. Still, we have chosen to include some parameters which seem likely to be relevant, to give the reader an idea of the needs of this specific application.

Information should then be handled per transaction in addition to the other levels,
as that is one of the important levels of granularity of such systems. The parameters of interest would typically include numbers of transaction coordinators and slaves per node, and the amount of relational algebra operations being evaluated. Memory usage should be shown specifically, showing the user for instance how much memory is used for various kinds of buffers, and for different operations, such as transactions, algebra operations, complex queries, and checkpointing. On the lower layers the programmer will want to have lock and log information and application specific communication information.
Chapter 13

Debugging Requirements

This chapter discusses what support the programmer needs for debugging our parallel computer, since it specifically addresses the HC16, some aspects and arguments will not be relevant on other parallel computers.

13.1 User Traces

One of the most common methods of debugging is to insert printf statements in the program, to see how the variables are altered through the execution of the program. The programmer usually prints out variables, i.e. shows some substate of the system, or text showing where processing takes place. In systems without specific debug support, this is often the only means available to the programmer for peeking into the computer, determining what goes on. This method has one very real advantage: it is simple, and can be used without learning any specialised debug language or worrying about alternative compilation and linking as is common with many debugging systems. Among the main draw-backs are the facts that altering the amount or selection of data involves recompilation of the program, and that it is often difficult to get an overview over the data, if significant amounts are generated. The numbers are often overwhelming, and would be more readable if presented graphically. We therefore want to add system calls which generate a plot of the users data on a graphic terminal. These can be enabled or disabled at compile-time. What the user chooses to put on such a display is her own choice. For example, in typical applications this might be "amount of work left" or "requests processed" or something of that kind. It should be possible to enable and disable this facility for one or all displays. This should take place in the nodes, at runtime, where it will stop all message generation.

As a next step we would like to let the programmer set up functions to be called when events occur, which should be able to evaluate program-specific aspects of the program state, and do traces on those. This evaluation could be sampling, i.e. at fixed time intervals, or event-driven, i.e. when a change takes place. The data could be saved, or the traces plotted interactively.
Another alternative would be to let the user specify variables or expressions to be monitored, and generate their plots automatically, updating the information on the screen each time the value changes or by polling. Statements to monitor variables only could have been inserted at compile-time, had it not been for the presence of pointers in the language. Since pointers exist, the way to implement this feature would be with hardware monitoring on the bus of the node, trigging when a specified address is updated or accessed. Expressions may be much more complicated, and require an arbitrary amount of evaluation.

Discrete values can be displayed per node, with histograms. They can also be displayed in various other ways, as discussed in Section 6.5. We shall not go into this, but at a later time an option of showing values from all nodes in one window should be considered. In particular we would like to see a MasPar-style visualiser\(^1\) [Mas90], showing a grid of the computer, marking the nodes which have fulfilled some condition. This would be especially useful for boolean or binary values, but we see situations where multi-valued displays could be of use, too. Due to limitations in time and scope of this work, we do not pursue this line of investigation.

### 13.2 Stopping Nodes

We want to be able to manually "freeze" nodes, i.e. halt all user-execution and make it look like time stands still on selected nodes. This is useful for testing the robustness of programs. The debug controller should provide a button to freeze nodes and show a bitmap of the nodes we have "frozen."

Later in this and the following sections we will discuss programmed debugging. Let us for clarity mention here that a node being stopped from the programmed interface will enter interactive mode. They do not get "frozen", going interactive means that they still handle interactive commands and queries. Frozen nodes will respond to queries until commanded to "thaw".

### 13.3 Inspection

When debugging a multicomputer, we frequently need to inspect the internal state of the system. One way of achieving this is to stop the computation\(^2\). We do not want our inspection to alter the path of execution, therefore we must commence execution with the same system state as when it was halted\(^3\). There are two intuitive ways of

---

\(^1\) The MasPar debugger gives the programmer the possibility of keeping a boolean matrix over all nodes updated, showing those where an expression evaluates to "True". This expression is limited to a comparison between a single variable and a fixed value.

\(^2\) State inquiries can also be directed to a database recording all intermediate states, post-mortem or "pipelined", as suggested by Mohr [Moh88]. See Section 9.3.1.1 for a closer discussion.

\(^3\) The system state is very complex, consisting not only of the values of user variables in the participating jobs, but also of communication processes and operating system primitives. A perfect
13.4 TRACING & REPLAY

achieving this: restoring the state after inspection or never destroying it in the first place. Event-based debuggers typically keep adding information about the state of computation, so that they can restore all or some aspects of the state at any or some time of a computation. If the entire multicomputer can be halted instantly, restoring is not needed to be able to continue computation. The state is intact and execution can be continued. Event tracing and storing is typically expensive (at least when done without added hardware) and will distort execution and reduce its efficiency, this problem is not present if stopping is based on instant halting. We have argued elsewhere that programs should be robust with respect to execution distortions. We would still like to minimise distortions, but since they will be present anyway, this is not a valid argument against event-tracing.

When inspecting the state of the computer we have opted for the freezing and continuing instead of tracing and restoring.

13.4 Tracing & Replay

As most programmers will surely agree, sporadically recurring bugs can be extremely difficult to correct. It is essential for the removal of such bugs that they are found, and finding them is tedious. One solution to this problem is to trace all events in a program, and save them. Then the same execution can be run again, and conditions can be recreated, allowing the bugcatcher to iteratively narrow in her search.

One approach taken is to use the stored events to replay individual processes, simulating all communication with other processes and devices. Then each process can be debugged on its own with a slightly extended traditional debugger. This is conceptually easy as the concurrent handling of several processes is removed. In many systems, all processes execute alone in their respective address space. Given this simplifying assumption, individual process replay may be acceptable, as processes are then somewhat "loosely connected" anyway. However, this approach would remove the possibility of seeing the cooperating processes as an integrated whole, and thus restrict the programmer in two ways. It would limit her insight into how concurrent parallel threads within one address space interact, and it would severely limit the possibility of looking at the state of the system as the sum of the state of several cooperating jobs in different address spaces, on different nodes. If the problem were to lie in the interaction between possibly remote jobs, it might be very difficult to discover within one job reexecuted on its own.

We think the integrated view of more than a single job is an important aspect of high-level debugging. We would therefore like to be able to rerun the entire computation, i.e. all jobs at once, and specify debug criteria over all of them.

---

Restoration is extremely difficult to achieve, since this involves not only software issues at all levels, but details of hardware, such as the position of disk-heads etc. The state must be similar enough to let continued execution make sense, the level is a trade-off issue, better state restoring will give better results, but at increasing cost.
As argued, although in a somewhat different context, by Aral and Gertner [AG89], a parallel computer can exploit parallel processors to implement costly debugging. In our case, some of the hypercube could be used for supporting the debugging of the rest. It would trace events during execution and regenerate them at replay time. Since our “boxes” are designed to be database computers anyway, they are ideally suited for storing the large amounts of data involved. In the HC computers, we would choose one dimension for splitting the cube into executing and supporting processors. The nodes on one side of this dimension would run the application program, and the nodes on the other side of that dimension would run a debug support program.

The communication across one dimension is fast, but not negligible. The storage of events would slow down and alter computation somewhat. Runs would be executed with a smaller number of processors, also altering execution in comparison with production versions of the application program. At this fairly low cost, the programmer would be offered determinismistic reexecutability; nothing less than a bargain!

Traces will be large. In some cases, large portions of them are irrelevant. The possibility of only saving when some condition is fulfilled should be investigated. The saving criteria must be well-defined and general. This would reduce storage requirements significantly, but also increase the need for re-execution if the programmer chooses unsuitable criteria in one run. Such re-executions are unfortunate, because races in the program may behave differently. The programmer would then have to build up a new understanding of the execution.

13.5 Debugging Performance

In a parallel computer, performance is seen to be an important issue. If performance is the motivation for using a parallel computer, low performance is critical. Performance bugs definitely have to be addressed, we think this is an issue for the debugger. Recognising this, we would also like to halt execution or take action depending not only on logical states, but also on the performance metrics of the computer. This information could be collected in the nodes, or in the integrated monitor. The use of this would be to inspect computation when odd performance is observed; the user might want to give commands like: “stop when computation load is less than x and communication load is higher than y.”

Monitoring will involve only a very small amount of extra work and intrusion on the nodes. The bugcatcher can sit and watch the monitoring displays, and then press the freeze button when detecting an unusual situation. The halting command has to be sent back to the running node, meaning that the stop will be delayed somewhat. When the nodes are stopped, the user can through the other debugging tools gain insight about the internals of the computer.

When problems are encountered using the performance debugging system, the bugcatcher needs a way of inspecting the computer, i.e. a debugger. When a state is recognised using a programmed debugger, this is also the case. We see that pro-
grammed debugging and performance debugging are different activities, but that both need traditional debugging support.

In Section 14.2 we discuss another way of debugging performance.

13.6 Process Integration

Traditional, sequential programs operate in one name space and one address space\(^4\). There is a single program counter, showing exactly and uniquely how far computation has come. Traditional debuggers offer the programmer the possibility of setting breakpoints in the source code, or more accurately, specified in the source code, but set at the corresponding place in the executable code.

What actions the programmer can request at these breakpoints depend on the quality and ambition-level of the system. In some systems the breakpoint can also be bound to some action, ranging in complexity and generality from halting to the printing of a variable, to the conditioned execution of dynamically loaded code.

13.6.1 Name Spaces

Typical MIMD systems consist of a set of independent nodes, and there is no notion of address-space encompassing more than one node. This means that there is no way from one node to access variables on a different nodes.

The individual nodes of a parallel computer cooperate to reach their common goal. Each node has its own part of the global state. In the sequential case, the entire state of the system was available within the debugger. For a traditional debugger this would not be so in a parallel system\(^5\). Hence, such systems do not allow specification of general criteria over the entire system state.

We therefore need a way of formalising criteria over a parallel computer as a whole, i.e. specifying over the entire state of the system. This means that we need to introduce a higher notion of address space, or to specify criteria over multiple address spaces. These solutions need not be very different, but we have chosen the latter.

13.6.2 Threads

Sequential systems run single threads. The breakpoint concept is well defined and well suited to debugging such systems.

\(^4\)In one name space, the global variables in the program are located at the same (relocatable) addresses. In one address space, all locations are directly addressable.

\(^5\)Traditional parallel debuggers in parallel system is here used according to the terminology introduced in Section 8.2. A debugger of this kind looks similar to a set of independent debuggers, one per node or process in use.
In SIMD systems, all processes execute the same code, and in lock-step. In this setting, all nodes reach the same place in the code at the same time, and the breakpoint concept mentioned above still makes sense. Admittably, specifying conditions will require more thought than in sequential systems, because variables in different nodes will have different values. By introducing some extra semantic, e.g. make breakpoints explicitly conditional on all or at least one node qualifying, conditional breakpoints again make sense.

MIMD computers are harder to handle. Independent processes and independent nodes will do different things, at wildly different positions. This is inherent in the definition of these systems.

We still want to be able to stop execution when interesting states are detected, hence we must define what we mean by a state. We must also be able to observe and specify these states. Debugging criteria evaluated on one node are limited to only a fraction of the system state: the local part of it.

Sometimes the internals of one node, the local state, is enough to specify our halting criteria. This is for instance the case if we want to halt when a variable is set to some specific variable, or when some values within the same address-space stand in some relation to one another.

In the general case, this is not so. An MIMD computer is an integrated set of computing resources; the entire state encompasses all the nodes. We need to be able to control execution based on the entire system state, i.e. all the nodes. This would for instance be necessary if we wanted to detect that a variable on one node had the same variable as some variable on another node, or more generally if the global state is such that interesting situations exist that are not detectable within one node.

We also want to evaluate our criteria without distorting execution. Distorted execution is unfortunate, since it means that different bugs appear when the debugger is in action, in comparison to when it is not.

### 13.6.3 Complex Actions

A parallel computer is a complex system, and debugging is tedious. It would be easier if we could automate some of the work. If breakpoints can only halt the system, real debugging involves a lot of manual work. The programmer has to check which breakpoint she hit, and then set up the system for continuing the work. A typical example of this is where the programmer knows that something odd goes on in one known function $a$, but only when it is called from another known function $b$. To investigate this problem, the programmer would either have to stop every time $a$ was called, and check if it was called from $b$, or it could stop every time $b$ was entered, and enable the breakpoint in $a$. This is overwhelming even in the sequential case; on a parallel computer, the number of possible situations is much larger. The programmer needs stronger support!
13.7 STEPPING

As we see, stopping in only one of the actions a programmer would like to have available when recognising some complex state. In general we would like to give the programmer the possibility of programming the debugger behaviour to examine and change states at whim, through the use of a “reasonable” programming language. We should give the programmer a programmable system, with states, variables and all the tools of the trade.

13.7 Stepping

Traditional debuggers often let the bugcatcher single-step the computer. This is interesting because it introduces the notion of a step, which is often one machine code instruction, or one line of source code. Larger steps can be achieved by setting explicit breakpoints where execution should halt, or by stepping over instead of into functions. If the bugcatcher steps into a function, many debuggers offer to continue execution until returning to the calling function. Splitting the code into functions reflects the programmers modularisation of the code, therefore it is useful to use this abstraction in the debugging process, too.

Some debuggers evaluate criteria after each step. This is not offered by all debuggers, many only evaluate when explicitly told to do so, or when hitting breakpoints. We suspect that the reason for this is that the cost of evaluation is so high that it is unattractive.

In a parallel computer, this situation only gets worse. The individual nodes will have to evaluate locally. To facilitate global criteria of even the simplest kind, global evaluation must also be facilitated.

One intuitive solution is to split a global criterion in local subcriteria, which could be shipped to the nodes. The nodes might continue executing until their local part of the criterion was satisfied. We see several complications with this scheme. If the bugcatcher wanted to stop all nodes when a special condition was detected in one node, only that one node would know that it should halt. Stopping the other nodes would require communication, and they would be executing too far before stopping. We call this the braking problem.

Also, the translation from a global criterion to a set of local criteria is not necessarily trivial, or even feasible. As we have argued above, the general case of evaluating a global criterion will in general require communication between nodes.

The braking problem can be solved by introducing lock-step execution. In asynchronous MIMD computers, it is unclear what to use as a step, but in order to do lock-stepping of the computer, we have to define it. Since communication is required between each step, the granularity must not be too small; if the steps are too large, the programmer will not be able to focus in as accurately as she probably wants to.

Timesteps can be physical, taking the same real time on all nodes, or logical. Since the programmers idea of time consumption is probably usually not very accurate, we
have chosen to use a logical step. We also wanted it to be fairly large to keep the overhead down, and to have some intuitive meaning to the programmer. We let a timestep\(^6\) be terminated when the user thread enters into or exits from functions. In addition to this, we introduce a special statement for terminating timesteps, called \textit{timestep}().

All breakpoints and criteria evaluation would now take place at the end of timesteps, i.e. each node computes through one timestep and then tells the controller that a timestep is finished. The controller collects information from all the nodes and decides what to do next.

This should be generalised to not only stopping, but also calling predefined or programmer-specified functions, which might in principle do anything, e.g. open a chart on the users instrumentation display and provide useful information, or allow for general programming.

This definition of a timestep implies that one timestep may be of different duration in each node. Indeed, any one timestep on the computer as a whole will have the duration of the longest local timestep, as it will effectively generate a synchronisation point in the execution of the program. Note that different nodes will be at different points in their execution.

This method is relevant for stepping the cube as a whole forward in time. The bugcatcher should however also be able to communicate independently with each node. Among the relevant actions to perform on each node is memory inspection and a traditional, local source-level or assembler-level step forward.

We think it is impractical to debug more than one \textit{job} at a time, since jobs are independent entities and observing several at the same time would add unneeded complexity. This decision will have to be reconsidered later. Threads of the same job, however, interact, and must definitely all be debuggable at the same time. Otherwise, the programmer would again be left with a restricted view of the system. The choice of which job to debug should be taken when a job is started, or during its execution. It is important that users be able to start debugging a job they thought could run without debugging.

Nodes will occasionally need to be stopped at different points of their execution, for instance if they satisfy their respective local criteria at different times. As an example, if every node is to halt when first invoking a function \(\Theta\), autonomy may let them enter at different times.

Sometimes we want to continue meaningful computation after inspection without restarting the system. If nodes stop at different times and start at the same time, a time difference \textit{will} remain, possibly altering the behaviour of the program unnecessarily.

\(^6\)In earlier versions, we used the term \textit{checkpoint} for the time when a timestep terminated, since this is the time when the criteria are "checked". The term \textit{checkpoints} are also used in other systems, it then means that the system makes a "dump", which can be restored in case of a crash or abortion. Unfortunately, it soon became apparent that these usages could be confused, so now we stick to \textit{timesteps}. 

sary. We need a way of synchronising the nodes. As a general way of doing this we
would like to be able to get the local time from all nodes, and to let all nodes run
past that time, until the next ending timestep. This could for instance be used to let
all nodes run as far as the current oldest node. The primitives should be general, i.e.
get a set of local times from the nodes, find the lowest, highest, or any specified time
in that set, and to let a set of nodes continue forwards to a specified time. Even though we currently consider a system without full event-tracing, the program-
mer might often find it useful to have access to information about the execution of the
debugger-script. When reaching a state, the bugcatcher will need to know how she got
there, i.e. the intermediate operations. Otherwise, debugging the debug script may
pose a problem in itself. We would like all requests from the debug-script together
with their results and all debug-script state transitions to be stored to a file made ac-
cessible to the bugcatcher. This will only involve local storing within the workstation
of a relatively small amount of information. A window continually updated should
be available for watching the contents of the nodeset variables of the debug-script
change.

Communication between the debug client and the nodes should come in two flavours:
automatic to any set of nodes using the debug language, and manually with any single
node. A “conversational interface” should be available, and might also need some
multi-node primitives, notably stopall, a command to stop all nodes currently running.
In case of individual debugging the active node could be selected with a button or a
command, and its identity should be apparent at all times. The programmed interface
will need boolean replies, meaning that all queries must be given a form where a
boolean answer is meaningful. As an example, the programmed interface will ask “is
thread x alive”. The conversational interface will be more user-friendly with verbose
replies, so that queries could be given a form more similar to: “which threads are
alive?”

13.8 Parts of the Debug Controller

As argued above, on option of deterministic replay adds clear benefits to interactive
debugging. As the tracing option will cost time and distort executions, a real-time
option is also necessary. The case where a program has been running unmonitored
for a long while and the programmer wants to know what is going on must also be
covered. In short, we need a trace/replay version in addition to a real-time version.

13.8.1 Instrumentation Unit

The instrumentation will give continuous information about the state of the nodes.
It will include a facility to stop the computer when certain criteria based on the

---

7In a system with a full event trace, earlier states could be deducted. We restrict time to forwards
movement, time warps as described in Section B.1 would confuse matters too much.
information available in this unit are met. It will display a set of graphics, including load meters, communication meters, a map of the active nodes etc. This should be offered both for play and replay. To make a compromise of all this, we would like to see the monitoring system provide:

1. Rough measurements at virtually no cost, this will give the user that nice feeling she usually gets when seeing the perf-meter or the disk-indicator flashing: “Ah, everything looks good, probably the computer is doing something useful”. This should give minimal invasion.

2. An alternative with a low degree of invasiveness. The instrumentor should be able to request fairly precise information about the state of each node, including entry and exit from functions and programmer-defined statistics. “Skip the debugger, and get the real numbers.”

The instrumentation unit give the programmer a way of interactively stopping nodes.

13.8.2 Machine-wide Programmed Debugger

The machine-wide debugging sees the computer as a whole, and uses the debug control language to control execution. This language is described in Chapter 14. It can evaluate expressions on sets of nodes, and in a limited way control their execution. Among other things, it can halt execution on a set of nodes. It will also include some evaluation of instrumentation data.

13.8.3 Single-process Replay

There should be a possibility of replaying just one job and simulate all external communication. Special care has to be taken for the case when several threads are executing in one address space. It will be similar to any normal, sequential debugger. It will allow inspection of variables, printing of the stack, breakpoints in the code and possibly variable manipulation. It should also let the programmer call functions in the context of the running job. This tool is only relevant for replay mode.

13.8.4 Interactive Oracle

When jobs are stopped, the bugcatcher will need to enquire about the state of computation, i.e. ask an oracle\(^8\). The facilities needed include those catered for in traditional sequential debuggers, with the possible exception of modifying actions. In addition, it should extend querying to sets of nodes, to allow the programmer not only to see if a

\(^8\) The term oracle is here intended to suggest guidance and knowledge, and has nothing to do with a database system of the same name
criteria is true on one node, but rather to see what nodes it is true on. The operations meaningful on the computer as a whole will not be the same as those meaningful on one node. It might, however, include things as “continue execution”, “proceed until time \( t \)”, etc.
Chapter 14

Debug Control Language

This chapter describes the language for controlling the machine-wide programmed debugger, as mentioned in Section 13.8.2. The aim of the machine-wide programmed debugger is to specify "situations" over the entire parallel computer, and have the debugger perform actions when these situations occur.

Our debug control language will offer the programmer repetitive evaluation of criteria and detection of sequences of states. We think it is important in a complex system that not only one specific state can be recognised, but that the debugger can be programmed in a general way to modify its behaviour according to the development over time.

The debug script language is similar to C, with some added functions and modifications for set and state handling.

Let us distinguish between the state of the computer, and that of the debugger. The state of the parallel computer is the concatenation of the local states of all nodes, i.e. it depends on the values of all the local memories, registers, and in some systems the disks. A debug state exists inside the debug controller, which resides on an arbitrarily chosen workstation somewhere in the network. The debug script contains explicit descriptions of criteria and actions, associated with identifiers\(^1\). The debugger is said to be in a state according to the name of the debug sub-script it is currently executing. One of the actions that can be taken is to enter a different debug state. On the more formal level, a debug state is determined by which part of the script the controller executes in, and the value of all of the internal debugger variables.

The controller lets each node proceed through one timestep and then evaluates the criteria specified in the debug script. This implies that action bound to a certain criterion will not be executed instantly when the criterion could be evaluated to "True", criterion evaluation only takes place at the end of each timestep. Note also that this may mean that conditions may be logically satisfied for a while within one timestep, without being discovered and if the condition is no longer satisfied at the

\(^1\)In the prototype version these identifiers are numbers. In a production version they should be logical identifiers.
end of the timestep, the situation will not be discovered and reacted upon. It is important that the bugcatcher is aware of this limitation.

To make it easier to program, the notion of synchronicity is supported in script programming, i.e. the script evaluates a criterion, and seemingly instantaneously gets the result.

14.1 Time

Time considerations, e.g. in the form of "timeouts", might offer a new dimension to this scheme, but we fear that this subject might be complex to handle, especially in case of a debugger that can stop sets of nodes\(^2\). Stopping a node involves halting the local clock, and for instance makes it unclear how to handle an timing constraint specified over two nodes, where one stops and the other does not? Also, in most systems the programmer or bugcatcher has only inaccurate knowledge of how much time each operation takes, making it difficult to specify meaningful time criteria.

Imagine also the case where the system is programmed to stop if first state \(A\) is detected, and then state \(B\) within time \(\delta\). Lets now assume that state \(A\) is detected at time \(T_1\), and then again at time \(T_2\), and state \(B\) at time \(T_3\). Now the time difference between \(T_1\) and \(T_3\) might be larger than \(\delta\), whereas the difference between \(T_2\) and \(T_3\) is not.\(^3\) The handling of this really turns fascinating (well, difficult) if you allow the programmer to consider not only maximum time between two events, but also minimum. Time considerations within the debugging process seems to be an interesting area for further research, but is considered to be outside the scope of this thesis. The approach described below gives the programmer some minimal tools for handling time in her programs, through reading and use of time in her programs.

14.2 Predicates

As a basis for reasoning about states in the parallel computer, we need to have a set of lowest-level predicates, from which we can build complex state descriptions. According to the discussion in Section 8.1, we have split those into execution predicates, data predicates, and instrumentation predicates.

The execution predicates regard the state of the program, we offer two of those:

**Called(funcname)**: this is true if a function is in the stack, but it need not be on top (or bottom) of the stack.

---

\(^2\) Bates and Wileden [BW83] argue that time on a parallel computer will be too skewed, no matter what algorithms are use to correct it.

\(^3\) This seems analogous to shift/reduce conflicts [ASU86].
14.2. PREDICATES

**Active(funcname):** this is true if the function is the executing one, i.e. on top of the stack.

In addition the bugcatcher may want to limit herself to one thread, or work on another thread, therefore these predicates are offered:

**Process(procname):** True if procname is the name of the executing thread.

**ChooseProc(procname):** This command returns true if it managed to change context to the named thread. It will not affect execution.

The only data predicate is:

**Eval(expression):** giving an expression as parameter, returning true when the expression is satisfied.

To achieve integration of debugging and performance debugging, we let the programmer reason about performance issues in the same ways as about data dependent predicates. The instrumentation predicates should provide access to all monitored parameters, as discussed in Section 12.3, in boolean and numeric form. The debug language only uses the boolean form, which looks like this:

**Monitor(metric, operator, value:** Metric indicates which metric the request concerns. Operator gives the choice of < or >\(^4\), and value is the value to compare with.

A quick example returning the set of nodes with more then five threads in the ready-queue:

**Monitor(cpu-load,>,5)**

The programmed debugging system described above would affect execution heavily. We have to admit that we are somewhat bewildered by the consequences of integrating performance debugging into the programmed debugging in this way. There are several reasons for this. The monitoring system gathers information at fixed intervals. Although the frequency can be modified, the granularity and "freshness" of data is important. Also, the stepping mechanism may in some cases distort execution heavily, resulting in numbers of little relevance. An alternative to our approach would be to offer a programmed interface to monitoring data only, and thus avoid the stepping. This would result in less invasion, but the integration of logical and instrumentation criteria would be more complicated.

\(^4\) Exact comparison of real numbers is better avoided.
14.3 NODESET Operations

In the debug control language, declared variables can be global to the script, or local to one state. The types include all the normal types of C, with the addition of a NODESET. ALL denotes the full nodeset and NONE the empty one. Below s denotes that a function returns a set, b a boolean, t a timeset, and i an integer. The following operations are defined:

\[ s = \text{SET\_ELEMENT}(x) \] with \( x \) of type integer sets \( y \) to a nodeset with just one element, the one denoted by \( x \).

\[ s = \text{SET\_OR}(\text{old,new}) \] sets \( s \) to the union of the two arguments.

\[ s = \text{SET\_AND}(\text{one,two}) \] returns the bitwise intersection of the arguments.

\[ s = \text{SET\_SUB}(\text{many,some}) \] returns the nodeset \textit{many} with any elements also occurring in \textit{some} turned off, i.e. \textit{many} \& \textit{not} \textit{some}

\[ s = \text{SET\_NOT}(x) \] returns the inverse of \( x \).

\[ s = \text{SET\_XOR}(x,y) \] returns the exclusive OR of its two argument sets, i.e. any bit set in one but not in the other.

\[ b = \text{EQUAL}(x,y) \] returns True if and only if the two nodesets are identical.

\[ b = \text{SUPER}(x,y) \] will return True if the two nodesets are equal or \( x \) is a superset of \( y \), False otherwise.

\[ b = \text{PURESUPER}(x,y) \] returns True if and only if \( x \) is a superset of \( y \).

\[ i = \text{COUNT}(\text{set}) \] returns the number of elements in \textit{set}.

\[ s = \text{SET\_EVAL}(\text{set, local expression}) \] returns the result of evaluating the local expression on each node.

\[ t = \text{TIME()} \] returns the a set consisting of the local time on each node.

14.4 Debug Script Operations

Only a few commands are new under execution of the debug script:

\texttt{STOP(nodeset)} takes nodes in nodeset out of the active set, i.e. halts them. These nodes can not be restarted from the debug-script, but go into interactive debugging.

\texttt{CONTINUE(nodeset)} lets nodes execute through one timestep. The default if no argument is specified is to let all nodes not \texttt{stop'ed} continue.
**NEXT(state)** tells the system which state to be in when after the next timestep.

**NOW(state)** to jump to another label and continue processing from there right away. This is in essence a *goto* within the debug controller\(^6\). We see that a NOW performs the same state change as NEXT, but without delay.

These commands give the programmer a possibility of going through a sequence of states. One way to exploit that is to recognise a sequence of events. In a comparison with the DISDEB system (see Section 9.3.9), we find that the specification of events is different, but that the idea that the programmer has to be able to specify sequences is essentially the same.

The debug control language allows the debugger to change state depending on criteria evaluated on the nodes and in the debug controller. Later, we want the language to extend this to allow a wide range of operations on the nodes, including operations like: call function, and trace variable (show \(x\) on display \(y\)). For now we will restrict ourselves to allowing the debug controller to those mentioned. When only some nodes are allowed to continue execution through another timestep, the rest are implicitly waiting, but they are not stopped. To stop, they have to be explicitly taken into interactive mode, otherwise they are just waiting and may continue execution at a later time if told to do so. We realise that the debugger would be more general if the bugcatcher was allowed to execute several debug scripts at the same time. We however fear that this would complicate the implementation, and leave that as a possible extension when our approach to debugging has shown itself useful.

We give nodesets as a parameter to all commands and actions, to direct where execution or evaluation is to take place. In the case of evaluations, the use of a limiting parameter will improve the performance somewhat, but readability will have more to gain from it. The same effect could, however, have been achieved by other set operations. As an example of this, imagine that we want to find the nodes satisfying \(\alpha\) and \(\beta\). This can be done by either finding the nodes satisfying each criterion, and finding the intersection of those, or by first finding the nodes satisfying \(\alpha\), and then querying about the nodes from that set also qualifying for \(\beta\). There is no semantic difference.

In the case of actions, nodeset parameters are very useful and necessary. The STOP command lets the programmer disable sets of nodes, and carry on debugging on the rest. The presence of a set argument to the CONTINUE command allows the programmer to let the rest of the nodes wait until a later time, and then possibly reactivate them. In the meantime, the nodes are still under control of the programmed debugging, and are as such not accessible for interactive debugging. This setwise control gives the programmer very detailed and explicit control over the system, and it also allows her to arbitrarily affect the relative advance of the nodes, she can for instance halt some nodes early on and let the others finish the entire program before those waiting are reawakened.

---

\(^6\)Not called so to avoid using a four-letter word.
At any time, requests evaluated in the debug kernels on each node will be independent of earlier requests. When the user wants to build up knowledge over time, this has to be done in the debug controller, through user-specified sets.

As one of the parameters to job and thread creation, the programmer should specify a name. If none is specified, a default should be supplied, to keep things tidy\(^6\). We solve nameclashes by appending numbers to the names, e.g. letting instances of a job called sorter be called sorter, sorter1, sorter2, etc.

Appendix 15 gives several examples of the use of this debug control language. Here is a trivial one where nodes trying to get into the function \textit{invertMatrix} are stopped:

\begin{verbatim}
NODESET mine, rest; /*ALL defined from system!*/
1:   rest = ALL; /* something like SET_ALL*/
    NOW(2); /* at once and later*/
2:   mine = SET_EVAL(rest,"Active(invertMatrix)"");
    stop(mine); /* this set of nodes into passive mode*/
    rest = SET_SUB(rest, mine); /* set for later checks*/
    CONTINUE(rest); /* all nodes not stopped continue*/
\end{verbatim}

We see that \textit{mine} and \textit{rest} are defined as nodeset variables. We use the set first for storing temporary results, and the latter to remember which nodes are still executing. In state one, the debugger just initialises datastructures, sets up the debugger to be in state two when called again, and instantly jumps to state two.

In state two, the first statement finds those nodes from those still executing, which are now entering function \textit{invertMatrix}. The next statement halts those nodes. The third line modifies the set keeping track of which nodes are still not stopped, and the last line tells those nodes to go on. At the end of the next timestep, the script process will be reawakened, and will start from the start of state two.

### 14.5 Implementation

A program written in the debug control language must be compiled or interpreted for execution. As we are strong believers in using standard tools, we have tried to find a way to minimise the effort required for this.

The syntax of the debug control language is a small extension of C. We therefore convert programs in the debug control language to equivalent C code first. This conversion should be done by a standard tool like \textit{awk} or \textit{Lex/YACC}. In the prototype version we use a small Emacs Lisp script for this conversion.

\(^6\)The programmer should blame herself if these default names are meaningless, and start using proper parameters!
If this conversion is successful, the program can be compiled to relocatable object code, using a standard C compiler. The resulting code will then be dynamically linked in to the running debugger. Currently this is be done by the tool DLD, [Ho90a, Ho90b], a library package that performs dynamic link editing (which we have ported to the local Unix version for this purpose). A pointer in the debugger will be set to point to a debugger function specified by the user, which is to be called at each breakpoint.

The sets of the debug language are trivially implemented by the standard type mechanisms in C, through the use of include-files. Most operations on sets can then be formulated as macros, and put in the headerfiles. Other operations will be compiled and loaded as libraries.

The debug language is executed as states. When all the nodes terminate a timestep, execution in the script process executes in such a state. Therefore we have to be able to start execution at the start of these state descriptions, and to terminate this execution. In C, pointers to functions are allowed, but not pointers into functions. We see that by implementing each state description as a C function, our calling requirements are satisfied. Therefore we convert each state in the debug language to a C function.

Since we have defined timesteps to terminate when functions are entered or left, the system needs to recognise those situations. We have chosen to achieve this by compiling in calls to the debug kernel on function entry and return.

To illustrate the conversion, we now show a debug control program to stop all nodes when all nodes having called msg_receive have msg_len of value zero, and the resulting C program:

```
NODESET receivers, new;

1:  
    receivers = NONE;
    NOW(2);

2:  
    new = SET_EVAL(ALL,"Called(msg_receive)") /* Add new callees */
    receivers = SET_OR(receivers,new);
    if(COUNT(receivers) > 0)
      NEXT(3); /* Now and until call of NOW <lower> */

3:  
    receivers = SET_OR(receivers,SET_EVAL(ALL,"Called(msg_receive)"))
    new = SET_EVAL(ALL, "msg_len==0");
    if(SUPER(new,receivers))
      stop(ALL);
```

---

7This is a general method, opening up for a later redefinition where only some of the these occurrences terminate timesteps, i.e. exactly those which the programmer chooses to compile in.
This should translate to the following C-program:

```c
#include <debug.h>
NODESET receivers, new, all;
extern void (*debugfunc)(); /*Initialised suitably*/

void debug1()
{
    receivers = NONE;
    debugfunc = debug2; /* NOW(2); */
    (*debugfunc)();
}

void debug2()
{
    new = SET_EVAL(ALL,"Called(msg_receive)") /* Add new callees */
    receivers = SET_OR(receivers,new);
    if(COUNT(receivers) > 0)
        debugfunc = debug3; /* NEXT(3); else unchanged */
    return;
}

void debug3()
{
    receivers = SET_OR(receivers,SET_EVAL(ALL,"Called(msg_receive)"))
    new = SET_EVAL(ALL, "msg_len==0");
    if(SUPER(new,receivers)){
        stop(ALL);
    }
}

We have assumed an external function pointer called debugfunc, whose contents are invoked on each sync-point. A NOW comprises an update of this variable and an immediate call of it, a NEXT an update and a return.
Chapter 15

Examples of Use of PanParallacea

This chapter contains several examples of how to program the debugger, of increasing complexity.

This program stops all nodes as soon as one node calls the function \textit{func}:

\begin{verbatim}
int c;
NODESET nodes;

1:
    nodes = SET_EVAL(ALL, "Active(func"));
c = COUNT(nodes);
if(c > 0)
    STOP(ALL);
\end{verbatim}

This is a different program doing exactly the same thing:

\begin{verbatim}
1:
    if(SET_EVAL(ALL, "Active(func")))
        STOP(ALL);
\end{verbatim}

The following program stops any node calling the function \textit{func}, but lets the rest continue:

\begin{verbatim}
NODESET stoppers;

1:
    stoppers = SET_EVAL(ALL, "Active(func"));
    STOP(stoppers);
    CONTINUE();
\end{verbatim}
Now for some more fun; we would like to wait until all nodes have been into function 
\(\text{func}\), and then stop all nodes when more than one node has entered function \(\text{next}\_\text{func}\):

\[
\text{NODESET funcers, nexters;}
\]

1:
\[
\text{funcers} = \text{NONE;}
\text{NOW(2);} \]

2:
\[
\text{funcers} = \text{SET\_OR( funcers, SET\_EVAL(ALL, "Active(func")\));}
\text{if(fancers == ALL)}{
    \text{nexters} = \text{NONE;}
    \text{NOW(3;)}
}
\text{CONTINUE();}
\]

3:
\[
\text{nexters} = \text{SET\_OR( nexters, SET\_EVAL(ALL, "Active(next_func")\));}
\text{if( (COUNT(nexters) >1)}
\text{STOP(ALL);}
\text{CONTINUE();}
\]

Imagine also that we, after having seen all nodes entering \(\text{func}\), want to stop all nodes,
when more than one node is in \(\text{next}\_\text{func}\) at the same time. We only need to alter one
line in the last part of the script:

3:
\[
\text{nexters} = \text{SET\_EVAL(ALL, "Called(func")\});} \quad /* \text{changed line */}
\text{if( (COUNT(nexters) >1)}
\text{STOP(ALL);}
\text{CONTINUE();}
\]

More complex demands can be readily met. The following script makes sure that
the nodes executing function \(f\) execute on their own, and that any functions calling
function \(a\) are stopped if they also have function \(b\) in their call stack:
NODESET set;

1: 
   if(set = SET_EVAL(ALL, "Active(f")}){
      CONTINUE(set);
   }else{
      set = SET_EVAL(ALL, "Active(a")};
      STOP(SET_EVAL(set, "Called(b")});
      CONTINUE();
   }

Now for some performance monitoring. The following program will stop processing if four or more nodes have a load over three, and at the same time four or more nodes have a load of less than one:

1:  if(COUNT(SET_EVAL(ALL, "Monitor(cpu_Load,>,3")}) >= 4) {
      if(COUNT(SET_EVAL(ALL, "Monitor(cpu_Load,<,1")}) >= 4) {
         STOP(ALL);
      }
   }
   CONTINUE();

Up to now, we have limited ourselves to simple sequences of debug-states. The following piece of code has a conditional branch. When all nodes have been in function send, it starts waiting for them all to get back out of that function, and then stops all nodes entering it again. If all the nodes are in the function send at the same time, it stops them as soon as they leave that function.
NODESET accu_send, left;

1:
   accu_send = NONE;
   left = ALL;
   NOW(2);

2:
   if((SET_EVAL(ALL, "Active(send"))) == ALL{
      NEXT(3);
   }else{
      accu_send = SET_OR(SET_EVAL(ALL, "Active(send")));
      if(accu_send == ALL)
         NEXT(4);
   }

3:
   STOP(SET_NOT(SET_EVAL(ALL, "Called(send"))));

4:
   left = SET_SUB(left, SET_NOT(ALL, "Called(send")));
   if(left == NONE)
      NOW(5);

5:
   STOP(SET_EVAL(ALL, "Active(send")));
Chapter 16

Conversational Debug Interface

The programming mode of the debugger will give the user the possibility of halting execution when certain criteria have been met. We hope that this will prove useful, however when any subset or all of the nodes have stopped, the user needs other tools.

There are in fact several reasons for having a conversational interface. When debugging, you first find the error, and only subsequently can you do something about it. When looking for it, you frequently do not know exactly what it does or when it occurs. The programmed mode of the debugger is based on the assumption that the programmer can specify suspicious states, or states where she is able to check if some aspects of the system behave as expected. When one of these states is reached, she will need the tools to probe around into all aspects of the global state.

When the sequence described in a debug script is recognised and all the actions are performed the debugger has to have some well-defined default action. One alternative would be to just release the execution, supported by the argument that the user should stop the nodes she wants stopped, the others must be unwanted. We think that this is more often destructive than not; and therefore choose to halt nodes by letting them “go interactive” when the debug script terminates. The user can leave debug mode by pressing a button, therefore this choice is a conservative one.

Ideally, the user should be able to use this interactive interface to ask questions about all the aspects of the state of single nodes or sets of nodes. If the “look-and-feel” of the interface is similar to that of the programmed interface, users familiar with that would be able to use it straight away. However, it is not obvious that the syntax of the relatively complex programmed interface makes it the easiest to use interactively. Also, some users might want interactive debugging without using the programmed debugging.

Ideally, the interactive interface should also give the user full read and update access to all data-structures and source code, and let the user call and jump as she might wish. Since one of the main purposes of a prototype of the suggested system is to demonstrate the usefulness of the programmed interface, we will try to limit and simplify the conversational interface severely, and therefore we have chosen to postpone
everything but queries in the interactive interface.

In this reduced interface, the commands will not be set-specific, and the results will be printed only, leaving this interface significantly smaller and easier to use (and implement) than the full-fledged programmed interface.

16.1 Commands

All commands consist of one line of input, returning results in what we have earlier termed "polite mode". Lines starting with number: are evaluated on node number only, all other nodes are executed on all nodes. These are the available commands:

called: Prints out the stack of called functions on a node. A synonym for this is where.

active: Prints out the active function on a node, also called in.

print var: Prints out the value of a variable. To specify a variable, it may be necessary to specify some subset of its thread, source file, function and name\(^1\). If the programmer specifies too small a subset, the debugger will ask for more information to be able to identify a variable uniquely.

dump address, count: show count bytes (default four) from the location address.

eval: expression: Evaluates the expression. It is yet unclear what general expressions the system will allow.

proc procname: without a parameter shows the default thread, with a parameter changes it.

list-proc: lists the threads, also stating their status.

time: returns local time.

goto-time: lets all nodes continue until the first timestep after time.

Note that all these commands are node local.

\(^1\)The set job, thread source file, function, and name is enough to uniquely identify any variable, apart from the case where the programmer introduces multiple block-local variables with the same name in different blocks inside one function. In this case the programmer deserves to be confused. Since we restrict ourselves to debugging one job whose name and number is known by the controller, the job name need not be specified.
Chapter 17

The User Interface

Figure 17.1 shows how a typical monitoring session looks to the user. The control panel is on the middle of the left side. Above that we see the number of threads in the ready-queue on each node, and on the top right the amount of milliseconds per 1006 millisecond period being used for user computation. The middle right graph shows the number of outgoing messages per second for two nodes. On the bottom right side we find the console of one of the nodes, all other consoles are closed. The bottom left view shows the disk activity, in this program there is none. This snapshot is taken during the second successive execution of a short program, that is the reason for the two peaks in communication and ready-queues.

In figure 17.2 we find a typical debugging situation. We recognise the control panel on the top left, the console of a node on the middle right, and the monitor display on the lower right. On the top right we find the host interface, mentioned earlier. The new parts are to be found in the lower left corner, first we see the debug control center, then comes a combined selector and display for showing which nodes are in interactive mode, and right from that the panel to freeze nodes. In this example, we see that one node is crashed.
Figure 17.1: A sample monitoring setup.
Figure 17.2: A sample debugging setup.
Chapter 18

Implementation Issues

As we want to combine debugging and performance debugging, we have chosen to integrate the debugging and instrumentation environments into one tool that both provides run-time statistics and debugging facilities. In some cases use of the debug facilities will distort the statistics, but in other cases the combined approach gives advantages justifying the integration.

We have implemented a prototype of the system. All major parts are ready, but there is still a small amount of integration to do before the implementation is complete.

18.1 Aspects Left Out

As we have seen, an ideal programming development system should consist of a large family of tools, like profiling, instrumentation, debugging, and replay. Since our time and resources are limited, we will have to restrict ourselves. In this section we will mention what we left out and how some of it could have been supported.

Profiling should be provided, giving the programmer a detailed summary of how and where time was consumed. This must be given on a per thread level, and tools should summarise this information into per-node and per-computer information. Implementing profiling is not too complicated. Most compilers already support options to generate extra code to store the needed information as computation goes along, this could trivially be kept on each node, although multiple threads might pose problems. Analysing the traces from each individual thread is a well-known technique, where tools already exist. Summarising traces will imply reading several such files and combining the information. It is also possible that all information needed could be extracted from an event-trace constructed for replay. Since our focus is elsewhere, we have chosen not to pursue profiling.

Replay is also essential, as argued in Section 13.4. Since we have primarily chosen to investigate multi-nodal criteria, we have found it necessary to restrict ourselves in other areas, and our experiments are based on “play” as opposed to “replay”
debugging. We think it should be feasible to extend the "play"-version to handle replay as well, once the appropriate events are available from the nodes. We hope to be able to look into this later.

The instrumentation/debugger processes could reside on different nodes from those being surveilled, i.e. it should be possible to allocate a set of nodes to run debug-processes for some other nodes. These processes would then have to communicate through "hooks" in the operating system on the nodes where the jobs being debugged reside. This would limit the intrusion on the active nodes.

The programmer should be able to define her own parameters for monitoring, either through providing functions for periodic sampling calls, variables for tracing, or choosing from a system-provided list of monitorable parameters. For now we limit ourselves to a set of standard parameters.

18.2 System Description

The debugging and monitoring system is involved in all parts of the computing system. It includes changes in the operating system, it requires a way to modify the execution on the node, its messages need to be transported through the host, and the debugging process will require a special tool for user interaction and control. The system requires communication between the nodes and the debug controller; responses and information flow out, commands and queries flow in.

Some basic instrumentation is always interesting, for instance to see that the system is alive. We have chosen to send out some monitoring data continuously, consisting of the local time of each node, the CPU utilisation, and some message statistics. It is possible to monitor other parameters too, but that information will only be collected and transmitted when the user requests it.

The system comes in several flavours:

- as an ever-present part of the operating system: rough measurements at virtually no cost,

- as a monitoring system with low intrusiveness, providing manual delayed halting of the system when certain performance conditions are fulfilled,

- as a debug kernel running on each node, providing inaccurate measurements but full debugging facilities, including source-level debugging of each node.

It is absolutely necessary that instrumentation and debugging can be invoked at any time, meaning that all supporting components must always be available and readily "wakeable". Restarting the system in debug mode is not an acceptable option.

Figure 18.1 shows how information and calls flow in the system. Arrows indicate calls, dotted lines bidirectional information flow. Each thread in the user process calls the
Figure 18.1: Calling and communication in PanParallacea
job-local debug kernel. This kernel switches between the address spaces of the kernel and the user, calling directly into the communication primitives without relinquishing control. All other communication goes through the communication process. Debug communication has high priority all the way through the system, to keep throughput acceptable.

18.3 Node

The individual nodes of the hypercube run TorOS. Some aspects of the operating system had to be modified to facilitate our debugging concepts. The implementation of the debugger was undertaken with the explicit goal of using the standard operating system, hence the modifications to TorOS had to be of such a nature that they would not disturb normal operation when not in use.

18.3.1 Normal Communication

As indicated in the figure, there is a communication process on the node. There is only one common communication process on each node, and it runs at a higher priority than the user jobs. This process will repeatedly try to send unsent messages, receive messages, and perform transit, i.e. forward messages going through the node. When the communication process has work outstanding which can not be finished at once, the reason must be that the neighbour-shared buffers are full if sending or empty if receiving. The communication process will then wait for an interrupt from the neighbour signalling that the blocking condition has been removed. Occasionally, the communication process prepares a timed wakeup call for itself and goes to sleep for a short while to avoid monopolising the CPU entirely. When there is no work to be done, it just puts itself to sleep indefinitely, awaiting interrupts. Normally, the communication process is awakened by interrupt from neighbouring nodes when there is a message coming in to the node, but if the communication process has more than a certain amount of work to do it will turn the awakening interrupts off and handle communication directly until the level of activity is reduced.

The communication is implemented as part of TorOS, although some extensions were necessary to support debugging.

18.3.1.1 Message Sending

When a user or system thread wants to send a message somewhere, a system call is performed. If there are no messages waiting to be sent over the dimension in question, the kernel attempts to send the message directly by calling into the lower layers of the communication system. In case of success, control is returned to the calling thread. If the message could not be sent or if there is already a queue for that path, the message must be queued up. The sending thread is then suspended until
the message is sent, or a specified timeout is reached. When a thread is suspended due to unfinished message handling, the appropriate communication process is set up to be awakened when the blocking condition has gone away. It will then try to handle all waiting messages, including the one directly causing activation, and reenter into the ready queue all threads thus released.

18.3.1.2 Message Reception

When a message is put into the DPRAM, the communication process on the receiving side is awakened by an interrupt. This interrupt may be disabled, the communication process then already knows that there is work waiting, and will inspect all in-boxes. When a message is found, it will copy the message onto the node, and inspect it. If the recipient is already waiting for a message, the message will be delivered and the recipient put in the ready queue. If the recipient is not waiting, the message will be kept and delivered instantly when the recipient issues its receive call. Internally, the communication is asynchronous, since sending and receiving is decoupled. The user interface is blocking, but has timeouts so that an error code is returned if the timeout is reached before the activity was completed.

18.3.2 Monitor Process

To support instrumentation, the node needs to collect data. We have chosen to do data collection by modifying library calls and some kernel routines. For instance, every time a message is sent, an accounting variable is updated. Information about things like CPU utilisation are collected by the scheduler. At regular intervals, the monitor process is awakened to build up a message containing instrumentation information, which it promptly sends out to the monitor display unit. These structures are reset when the information is transmitted.

Messages to modify instrumentation behaviour are also handled by the monitor process, it is activated and does the appropriate alterations as soon as such a message is received.

The alterations available from the monitor process include:

- Alter interrupt frequency.
- Check and update status and mode, and the list of monitored parameters.

We would like to include an option to add interrupt functions to be executed at frequent intervals, but the consequences and usability of this are not yet thought through.

The monitor messages are self-defining, i.e. they contain a map describing the rest of the contents. Their content and the frequency of sending can be altered by sending
messages to the monitor process. The user can also choose between seeing all activity or user activity only.

18.3.3 Debug Process

The debug process supervises debugging on the nodes. It switches debugging mode on or off, and informs the controller of the name and locality of jobs. It does not take part in the individual communication between threads and the controller, that is all handled by the debug kernel. The debug process activated by messages, and otherwise idle.

The system gives the user the possibility of freezing nodes. This can be used by the programmer to observe how her programs behave when some nodes are subject to arbitrary delays. Freezing is implemented by sending a message to the debug process. When receiving such a message, the debug process monopolises the node completely until it gets a message telling it to return to normal execution. This feature can also be useful when the programmer needs a breathing space, all nodes can be almost instantly frozen, and the programmer can look at output of various kinds (or get herself a cup of coffee) before allowing computation to continue as if nothing happened. When debugging is active, the freezing feature is not necessary, and the system can be halted after timesteps, but freezing is independent of timesteps, and can be used whether programmed debugging is turned off or on.

While nodes are frozen, the debug process takes over the role of the debug kernel in interactive mode, i.e. it evaluates requests and executes commands. The commands are the same as in stopped mode, but the bugcatcher has to specify job and thread in addition to the other parameters, since those are not here given by context.

18.3.4 Debug Kernel

The node's side of the programmed debugging is activated from operating system traps in the users code, and executes on behalf of the user process.

Each thread occasionally terminates timesteps, currently when calling and returning from functions, and when the programmer has explicitly programmed a timestep termination. After a timestep, the user thread calls into the kernel, where a job-specific debug kernel takes over control. If debugging is disabled, control is returned immediately.

Otherwise the debug kernel communicates with the controller, informing it of its status, without relinquishing control of the CPU. Local time is halted prior to this.

The normal communication primitives are too general for this purpose. We do not want the operation of the debug kernel to cause scheduling or to interfere with the behaviour of the system. The normal message sending may result in scheduling if a message can not be sent immediately; this disturbs execution too much to be allowed.
Instead, the debug kernel takes over control over the communication. First, it empties the queue over the host dimension. Since normal programs mainly communicate within the computer, this communication path is normally only used for systems work\textsuperscript{1}, and should be empty. It then "manually" drives all communication over that dimension to be able to communicate with the host. During the time of this exchange all other activity on the node is frozen, including the clock.

The debug kernel informs the controller of its status and location in the code. The controller then issues commands and requests, as described below. The requests concern the state of the active or other threads, and the commands may tell the debug kernel to modify datastructures or possibly execution. Eventually the debug kernel may be told to take the node into interactive mode, where the debug script is not used, and all subsequent actions depend on the bugcatcher's interactive requests. The debug kernel may also be told to continue execution, in case control is returned to the user thread. The user thread should not be able to notice that anything happened during the system call, nor the delay.

The debug kernel knows nothing of the mode of the controller. All queries from programmed mode generate boolean answers, typically to queries of the form "is this thread in function $x$?" Interactive queries are of the form "what function is this thread in?"

Figure 18.2 shows the setup and indicates how the debugging works on each node. The job-local debug kernel will given control at the end of each timestep, which may be much more frequent than the scheduling. It then stops all other activity, communicates with the controller, and eventually returns control to the thread from which it was invoked. Local time is stopped during this session; the executing thread should not be able to notice that the debugger was invoked, or that time passed before control was returned from the call.

### 18.3.4.1 Debug Kernel Operations

The debug language allows the programmer to reason about the state of the multi-computer. This must be supported by evaluations on each node. The debug kernel operations must support these operations in two forms, which we call boolean and textual. The boolean version is used for programmed debugging.

The boolean version has extra parameters, and returns True or False according to the evaluation result. The textual version returns accurate values. As an example take the two interfaces to the function requesting the active function, the first gives a boolean reply, the other returns the function name:

```c
int BActive(char* funcname);
char* Active();
```

\textsuperscript{1}Program loading is the main use of this channel, and that will normally not be overlapped with debugging.